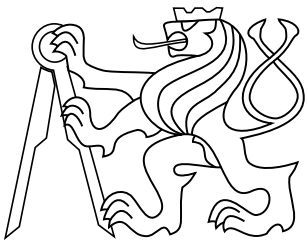
Master's thesis

Intent Detection Module for a Conversational Assistant

Daria Ozerova



May 2022

Supervisor: Ing. Jiří Spilka, Ph.D.

Supervisor on behalf of CTU FEE: Ing. Petr Pošík, Ph.D.

Czech Technical University in Prague
Faculty of Electrical Engineering, Department of Cybernetics

MASTER'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Ozerova Daria** Personal ID number: **466363**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Cybernetics and Robotics**

Branch of study: **Cybernetics and Robotics**

II. Master's thesis details

Master's thesis title in English:

Intent Detection Module for a Conversational Assistant

Master's thesis title in Czech:

Detekce úmyslu pro konverzačního asistenta

Guidelines:

1. Research the state-of-the-art sentence embedding models (sentence transformers).
2. Research data sets for finetuning of models for a sentence similarity task.
3. Analyze performance (accuracy, training speed, inference speed, and hardware requirements) for selected datasets and models.
4. Use a chosen model and implement an intent recognition module for a conversational agent.
5. Analyze model deployment strategy (serving) considering machine learning life-cycle.

Bibliography / sources:

[1] Vaswani, Ashish, et al. "Attention Is All You Need." Advances in Neural Information Processing Systems, vol. 30, Curran Associates, Inc., 2017. Neural Information Processing System.
[2] Devlin, Jacob, et al. "BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding." ArXiv:1810.04805 [Cs], May 2019. arXiv.org, http://arxiv.org/abs/1810.04805.
[3] Reimers, Nils, and Iryna Gurevych. "Sentence-BERT: Sentence Embeddings Using Siamese BERT-Networks." Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP), Association for Computational Linguistics, 2019, pp. 3980–90.
[4] Rogers, Anna, et al. "A Primer in BERTology: What We Know about How BERT Works." ArXiv:2002.12327 [Cs], Nov. 2020. arXiv.org, http://arxiv.org/abs/2002.12327.

Name and workplace of master's thesis supervisor:

Ing. Jiří Spilka, Ph.D. Concerto AI Czechia, s.r.o., Praha

Name and workplace of second master's thesis supervisor or consultant:

Ing. Petr Pošík, Ph.D. Department of Cybernetics FEE

Date of master's thesis assignment: **15.12.2022** Deadline for master's thesis submission: **20.05.2022**

Assignment valid until: **30.09.2023**

Ing. Jiří Spilka, Ph.D. Supervisor's signature	prof. Ing. Tomáš Svoboda, Ph.D. Head of department's signature	prof. Mgr. Petr Páta, Ph.D. Dean's signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce her thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

Acknowledgement

I am very grateful to my supervisor, Jiří Spilka, for his guidance and suggestions, and I value the experience and knowledge I have acquired when working on this thesis. I thank Petr Pošík for his assistance on behalf of FEE CTU. I am grateful to my colleagues, Václav Chudáček and Tomáš Brich, for their valuable remarks and feedback. My biggest thanks belong to my family and friends for all the encouragement they gave me throughout the years. I am particularly grateful to Vítek for his love, patience, and support.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Date

Signature

Abstrakt

Konverzační asistenti a chatboti se stali oblíbenými nástroji, které rychle a snadno odpovídají na dotazy uživatelů a pomáhají jim řešit jednoduché úkoly. Jedním z nejdůležitějších úkolů konverzačního asistenta je porozumění záměru uživatele. Nedávné pokroky ve zpracování přirozeného jazyka zaznamenaly výrazného zlepšení přesnosti rozpoznávání úmyslů.

Tato práce si klade za cíl navrhnout a implementovat systém rozpoznávání úmyslu, ke kterému lze přistupovat prostřednictvím aplikačního programovacího rozhraní. Výsledkem práce je modul pro detekci záměrů, který je schopen klasifikovat zprávy od uživatele z předem definované množiny záměrů. Využívá nejmodernější modely založené na transformerech MiniLM a Robustly Optimized BERT Pretraining Approach (ROBERTA) v kombinaci s neuronovou sítí. Tento přístup dosahuje velmi dobré přesnosti na srovnávacích datasetech (96% na SNIPS a 96% na CLINC150 in-scope). Na menším, interním datasetu, je výsledná přesnost 88%. Nejlepší model dosahuje přibližně o 10-15% lepší přesnosti než řešení, které se v současné době používá ve firmě zadávající práci.

Klíčová slova

detekce úmyslu, sémantická podobnost vět, klasifikace textu, embeddings, transformery

Abstract

Conversational assistants and chatbots have become popular tools that can quickly and easily answer users' questions and help them solve simple tasks. One of the most important goals of a conversational assistant is to understand a user's intent. Recent advances in natural language processing made significant progress in the accuracy of intent recognition.

This thesis aims to design and implement an intent recognition system that can be accessed via an application programming interface. The resulting module can classify user utterances using labels from a predefined set of intents. It uses state-of-the-art transformer-based models MiniLM and a Robustly Optimized BERT Pretraining Approach (ROBERTA) combined with a neural network-based classifier. The approach achieves good accuracy on benchmark datasets (96% on SNIPS and 96% on the CLINC150 in-scope dataset). On the smaller, internal dataset, the resulting accuracy is 88%. The best model achieves approximately 10-15% better accuracy than the solution currently used by the company which supervised the thesis.

Keywords

intent detection, semantic sentence similarity, text classification, embeddings, transformers

Contents

Acronyms

AI Artificial Intelligence.
API Application Programming Interface.
AWS Amazon Web Services.

BERT Bidirectional Encoder Representations from Transformers.
BPE Byte-Pair Encoding.
BRNN Bidirectional Recurrent Neural Networks.

CBOW Continuous Bag of Words.

EC2 Elastic Compute Cloud.
ECR Elastic Container Registry.

GRU Gated Recurrent Unit.

LSTM Long Short Term Memory.

MAE Mean Absolute Error.
MLM Masked Language Modeling.

NLP Natural Language Processing.
NSP Next Sentence Prediction.

QA Question Answering.

ReLU Rectified Linear Unit.
REST Representational State Transfer.
RNN Recurrent Neural Network.
ROBERTA a Robustly Optimized BERT pre-training Approach.

S-ROBERTA Sentence-RoBERTa.
S3 Simple Storage Service.
STS Semantic Textual Similarity.
STSB STS Benchmark.

TPU Tensor Processing Unit.

UI User Interface.

Chapter 1

Introduction

Natural Language Processing (NLP) is a subfield of computer science that aims to understand human language. Its goal is to create a system that can analyze and process spoken or written words and texts. NLP includes machine translation, automatic text summarization, named entity recognition, natural language generation, and many other tasks. One of the most popular applications in NLP is a conversational assistant (chatbot), which is a software system intended to converse with humans.

One of the essential needs in a conversation between humans and Artificial Intelligence (AI) is understanding the user's goal. If a conversational assistant understands what the user wants, it can make competent decisions and provide the best possible service. Therefore, every intelligent conversational system implements a so-called *intent detection module*.

Intent detection (also *intent recognition*) is a common NLP task that deals with the classification of user utterances. It aims to classify a text using a predefined label called *intent*. The intent detection module illustrated in Figure 1.1 maps a user message to a set of topics that a conversational system can understand.

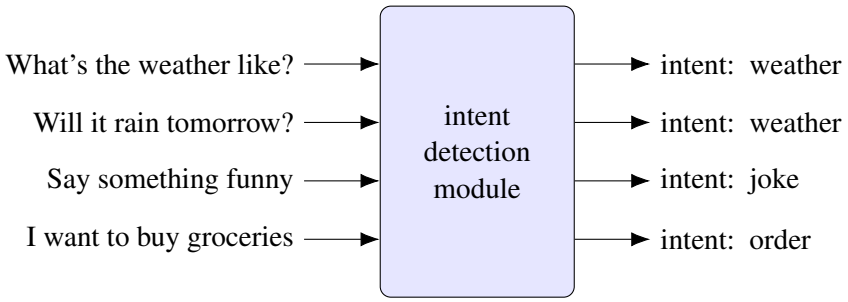


Figure 1.1 An illustration of the intent detection module's functionality.

Human language is complex, and people can describe the same idea by rephrasing sentences and expressions. Because of that, each intent can have a significant amount of different variations (also referred to as *templates*). Some can be predefined, but it is hard to capture all the possibilities. Furthermore, some utterances belonging to the same intent can contain entities, such as emails, phone numbers, or names, that can assume many unique forms. Defining all variations by hand is infeasible, and, therefore, an intelligent intent detection module is needed, so it can recognize a topic without having exact templates.

Historical intent recognition systems rely on symbolic methods that depend on syntax analysis and handwritten rules [1]. Later, statistical methods replaced them by introducing machine learning methods that automatically learn formerly hand-coded rules. It significantly improved the quality of intent classifiers [2]. Nevertheless, statistical algorithms, such as decision trees, require elaborated feature engineering to be able to transform sentences into vectors contain-

ing their attributes. Because of that, modern intent recognition systems shifted towards deep learning-based algorithms. They also require a numerical input, but the feature design is primarily automatic.

Many algorithms can transform sentences into vectors. The most useful are those which can capture an original text's meaning. These algorithms significantly increase the performance of intent recognition systems [3]. When numerical form captures the meaning, similar sentences typically obtain similar transformations. So when used as a part of intent detection, the module does not require a lot of predefined intent variations. Modern state-of-the-art encoding algorithms are Bidirectional Encoder Representations from Transformers (BERT) and its derivatives. These transformer-based models achieve the best results on various tasks [4].

A typical natural language understanding system consists of intent detection and *semantic slot filling* (also *entity extraction*) components [3]. An intent represents the general meaning of an utterance, and an *entity* is a specific piece of information for a given topic. Recognizing the correct intent is essential for detecting proper entities because different topics typically require the extraction of different entities.

1.1 Aim of this work

This work aims to create an intent detection module able to assign intents to user utterances. More detailed goals are summarized in the following paragraphs:

- **Application of state-of-the-art, transformer-based embedding algorithms**. An essential part of the intent detection module is an algorithm that transforms sentences into vectors. Similar sentences have similar transformations, so the quality of the encoding dictates the performance on the semantic textual similarity task. Therefore, a big part of this work is dedicated to semantic textual similarity and evaluation of different encoding algorithms.

- **Classification of an input text with reasonable accuracy**. A vital component of the module is an intent classifier, which maps a sentence's numerical representation to an intent. The most desirable outcome of a classification task is high accuracy. In this work, two approaches to intent classification are developed and compared. The first approach builds on semantic sentence similarity. It compares user utterances with a set of predefined templates belonging to a particular class. The second one is a neural network-based classifier. It trains on a predefined set of variations and predicts a label for a user utterance.

- **Low latency and memory efficiency.** In a human-AI conversation, a bot needs to act fast. It is not acceptable when a conversational assistant processes a sentence several seconds before responding. The inference time matters, so it is crucial to create a module's architecture that is robust, efficient, and can react quickly. A considerable part of this work focuses on designing and evaluating architecture that is acceptable for use in real applications.

- **Implementation of an Application Programming Interface (API) for access by other systems**. In order to allow access to other systems, the module will be deployed to a cloud environment. A large part of this work discusses and analyzes strategies for deploying machine learning-based components in production. The goal is to find the most efficient and robust but, at the same time, cost-efficient strategy.

This work is a part of NLP research conducted by the company Concerto AI, which develops automated conversation platforms. My work aims to propose a new design of an intent recognition module that incorporates state-of-the-art encoding models.

1.2 Structure of the thesis

This work is divided into six chapters, which describe parts of the intent detection module. Each part provides a general background, methods, evaluation, and results. Chapter 2 focuses on the embedding algorithms and semantic textual similarity. Then it explains and evaluates two encoding models on a standard dataset and investigates their usage in real applications. A considerable part of the chapter is dedicated to models' fine-tuning on the author's dataset, consisting of examples with unusual vocabulary. Chapter 3 explains the intent classification by introducing semantic similarity and neural network-based algorithms. It analyses their main properties and compares their training and inference speed and performance. The chapter also examines the advantages of the proposed solution over a current solution used in the company. Chapter 4 describes the machine learning life cycle and how all its parts are managed in the intent detection module. Chapter 5 explains the module's architecture and implementation. It also describes an applied deployment strategy and usage in real applications. Finally, Chapter 6 discusses the results and concludes the work. The whole work, including the experiments, is available at GitHub[1] with a Readme.

[1] https://github.com/ozerodar/intent_recognition

Chapter 2

Semantic textual similarity

Every machine learning algorithm processes texts in a numerical form. There are many ways of transforming sentences into vectors, but the most promising are those that generate vectors with the following properties:

- Transformations of similar words, expressions, and sentences are close to each other in a vector space.
- Vectors are dense, and every bit carries relevant information.
- Representations preserve the semantic meaning.

Many algorithms satisfy all of these properties, but some are more suitable than others for intent detection. Their quality is usually tested using the Semantic Textual Similarity (STS) task that aims to measure the relationships between words, sentences, and documents. STS is a regression task that inputs two texts and returns a real number representing their similarity. The performance on the task strongly relies on the quality of the encoding algorithm and is usually evaluated on benchmark datasets, such as STS Benchmark (STSB)[1].

The first part of the chapter explains the concepts of word and sentence embeddings. Then it discusses the metrics which allow measuring the relationship between sentences or texts. The third part focuses on the evolution of embedding algorithms. It analyzes the advantages and disadvantages of different methods. The rest of the chapter focuses on the state-of-the-art architecture called *Transformer* and explains how it can be used in real applications.

2.1 Word and sentence embeddings

One of the simplest ways to create a vector from a sentence is using a *one-hot* representation. Any document contains a finite number of unique words which compose a vocabulary. Any word $\mathbf{w}$ in a vocabulary can be represented as:

$$\mathbf{w} = \begin{bmatrix} 0 & \dots & 0 & 1 & 0 & \dots & 0 \end{bmatrix}^\top. \tag{1}$$

It contains one in the i-th row and zeros everywhere else. The value of i is equal to the index of a word in the vocabulary. A vector's features represent the presence of a word in a vector and the absence of any other words.

[1] https://ixa2.si.ehu.eus/stswiki/index.php/STSbenchmark

The whole vocabulary can be represented as:

$$W = \left\{ \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \ldots, \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} \right\}. \tag{2}$$

The dimension of each vector is equal to the vocabulary size. A vector representation of a sentence is a sum of vectors of individual words. The one-hot vector representation is simple and does not require any additional learning, but it has several disadvantages:

- **Vectors are high dimensional.** A document can contain thousands of unique words. Each of them will have a size of a vocabulary. It takes up a lot of memory and makes the computation slow.
- **Vectors are sparse.** A vector representation of a k-word long sentence has up to k elements that contain valuable information. The rest of the vector's elements are zeros. They represent the absence of every word from vocabulary except those contained in a sentence.
- **Vectors do not carry any semantic information**. It is almost impossible to compare words and sentences using this kind of representation because all words are perpendicular to each other. Based on this approach, the relationship between sentences *"How are you"* and *"What's up"* is not any different from the relationship between *"How are you"* and *"I like pineapples."* Sentences can only be compared when they share some words.

Learned vector representation called *embedding* solves all the abovementioned issues. Embeddings are dense vectors that capture the meaning of words or sentences. They usually have lower dimensions, and each dimension can represent a unique feature (which is often hard to interpret). Similar texts usually have similar semantic features, so their vectors are close to each other in a vector space. Moreover, embeddings can capture different relationships between texts, like comparatives-superlatives or singular-plural [5]. These relationships can be helpful when dealing with sentence similarity and intent detection.

There are two types of embeddings. *Static* embeddings are representations that do not depend on the surrounding context. These vectors typically cannot capture multiple meanings of words. On the other hand, *dynamic* (or *contextualized*) embeddings always depend on the surrounding text, so the same word in a different context is considered as two unrelated words. In most cases, contextualized embeddings are more useful because many languages contain words having multiple meanings, and most of them need a context to be understood.

Embedding is a learned vector representation, so to be able to transform sentences into vectors, machine learning algorithms must use data to train their models. There are several machine learning-based approaches to embedding construction. Each of them has multiple variations, but only the main ones will be discussed in this chapter.

2.1.1 Semantic similarity measures

- **Euclidean distance**. One of the most intuitive ways to measure the relationship between vectors is Euclidean distance. Euclidean distance $d(\mathbf{v}, \mathbf{w})$ of two n-dimensional embeddings $\mathbf{v}$ and $\mathbf{w}$ is defined as:

$$d(\mathbf{v}, \mathbf{w}) = \sqrt{\sum_{i=1}^{n} (v_i - w_i)^2}. \tag{3}$$

The more similar the vectors are, the closer they are to each other in a feature space. However, some vectors can be similar in terms of orientation, and Euclidean distance does not consider it.

- **The dot product**. Similar words and sentences often have similar directions of their respective embeddings, so using the dot product is another intuitive way to measure similarity. The dot product $dot(\mathbf{v}, \mathbf{w})$ of two n-dimensional embeddings $\mathbf{v}$ and $\mathbf{w}$ is defined as:

$$dot(\mathbf{v}, \mathbf{w}) = \mathbf{v} \cdot \mathbf{w} = \sum_{i=1}^{n} v_i w_i. \tag{4}$$

The main disadvantage of using the dot product is that vectors with larger magnitudes will tend to have higher dot products even if they are not similar [6].

- **Cosine Similarity** is one of the most popular similarity metrics in NLP. It considers only vectors' orientation and does not depend on their magnitude. Cosine similarity score $s(\mathbf{v}, \mathbf{w})$ is equal to the cosine of the angle θ between $\mathbf{v}$ and $\mathbf{w}$, which is equal to the normalized dot product:

$$s(\mathbf{v}, \mathbf{w}) = \cos\theta = \frac{\mathbf{v} \cdot \mathbf{w}}{\|\mathbf{v}\| \|\mathbf{w}\|}. \tag{5}$$

The lower the angle θ between $\mathbf{v}$ and $\mathbf{w}$, the higher the similarity. Usage of cosine similarity solves the issues of the two previously mentioned metrics. Thus the rest of the work uses the cosine score as the primary similarity measure.

2.1.2 Word2Vec

Word2Vec is one of the simplest neural network-based embedding algorithms. It is a two-layer feed-forward neural network trained to reconstruct a context of words [7]. Figure 2.1 shows two variations of the algorithm:

- *Continuous Bag of Words (CBOW)* learns to predict a target word based on its context.
- *Skip-gram* has similar architecture to CBOW's, but it inputs a target word and predicts a surrounding context.

Both models, Skip-gram and CBOW, try to map inputs to the outputs. The embeddings of target words are calculated using the weights of the hidden layer. Similar words are often found in the same context, so their weights are updated in the same way, giving similar embeddings.

Word2Vec is an algorithm that generates word embeddings. Sentence embeddings are created by averaging vector representations of words contained in a sentence. Several other algorithms use similar principles as Word2Vec, for example:

- *Sent2Vec* [8] applies the same idea, but instead of a fixed-size context of a word, it processes the whole sentence. It also allows the encoding of short sequences of words (*n-grams*).
- An algorithm called *FastText* also uses a similar architecture but treats a word as a sequence of subwords (characters n-grams) [9]. The method allows a better representation of rare words because their subwords are often present in other words.

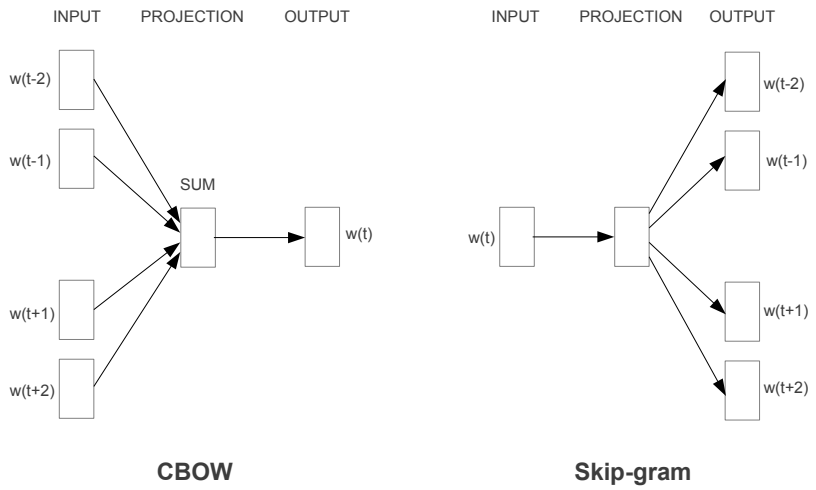


Figure 2.1 CBOW and Skip-gram architectures [7].

All of these algorithms have several disadvantages:

- **The inability to capture multiple meanings of words.** Most languages, including English, have words with multiple meanings. Some of them can even belong to different parts of speech. For example, the word *"back"* can be a noun (part of a human body), adverb (traveling in the opposite direction), verb (be in favor of), adjective (at the back of an object), and a part of phrasal verbs. Embeddings of these kinds of words are created using different contexts, so the resulting vectors are inaccurate.
- **Loss of word order**. Sentence embeddings are created by combining embeddings of words (sequences of words) contained in a sentence. Some sequences, for example, *"Were you at the museum"* and *"You were at the museum,"* will have identical embeddings, but their meaning is slightly different (one is a question, the other is not).

2.1.3 Recurrent Neural Network-based algorithms

A *Recurrent Neural Network (RNN)* contains a chain of several neural networks, and each of them processes a piece of data. RNN is a suitable model for processing data sequences, such as sentences.

RNN-based models used in NLP consist of the *encoder* and *decoder* [10]. The encoder generates sentence embedding by processing it word by word in a loop. Each network processes the output of the previous one and propagates it further. The last neural network passes a resulting vector to the decoder. A form of decoder depends on the task and can output, for example, a label or a sequence of data. For instance, for the machine translation task, it generates a sentence in another language, and for text classification, it can return a label using the so-called *many-to-one* architecture, illustrated in Figure 2.2.

There are several popular RNN-based models used for sentence embeddings and in NLP in general:

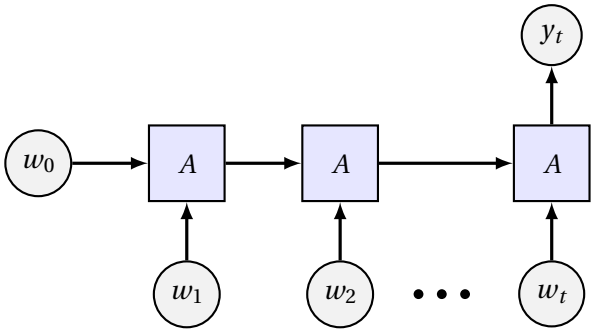


Figure 2.2 The *many-to-one* architecture used for classifying a $t-$word long sentence. The word w_0 is usually represented by zeros.

- *Bidirectional Recurrent Neural Networks (BRNN)* [11] consist of two independent RNNs. The first RNN processes the original sentence, and the second one processes it in reverse. They provide backward and forward context at each time step. The usage of BRNN can improve the quality of sentence embeddings because some sentences contain crucial information which is not "seen" by the model when processed from left to right. For example, the sequence *"I live in new"* can continue with *"York," "Jersey," "Zealand,"* and other terms. Backward context information can improve the quality of embedding in such cases.

- *Long Short Term Memory (LSTM)* [12] is a type of RNN capable of learning long-term dependencies. It introduces a cell state which remembers important information for a long time. Specific structures, called *gates*, decide when to add or remove information from a cell. In theory, RNN can process long-term dependencies. In practice, the training phase is problematic because of the vanishing gradient problem [12]. LSTM overcame this issue and can learn faster than classical RNN.

- *Gated Recurrent Unit (GRU)* is considered a variation of LSTM. It has similar but simplified architecture with fewer parameters, which is generally easier to train than LSTM [13].

RNN-based models are suitable for many NLP tasks. Nevertheless, they have several significant disadvantages:

- **Difficulty capturing the meaning of long texts**. Even RNN variations like LSTM, or GRU, have sequential processing limitations [14]. An input can consist of dozens of words, but the encoded text is transferred to the decoder only by the intermediate state. It is pressured to capture all the relevant information. Adding additional memory cells is sometimes insufficient to learn the true meaning of an input text.

- **Hardware under-utilization**. Due to sequential computation, it is challenging to process RNNs on GPU or Tensor Processing Unit (TPU). Hardware capacity is not used efficiently, making training and inference phases slower [15].

2.1.4 Transformers

To address some of the issues of RNN, researchers introduced a mechanism called *attention* [16]. It takes outputs from all hidden states and provides their weighted sum directly to the decoder. Weights are calculated in a way so that the more important a word is to a text, the more weight it has. These weights are learned automatically during the training phase. The attention

mechanism significantly improved the performance of systems that use the encoder-decoder architecture [17].

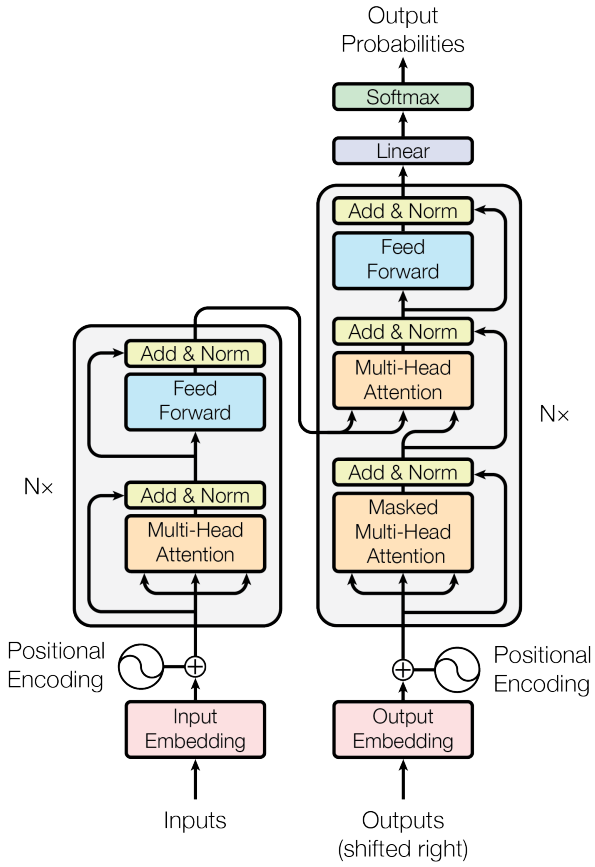


Figure 2.3 The Transformer architecture [18].

In [18], researchers proved that the attention mechanism used in RNN provides better embeddings than RNN itself. They introduced the *Transformer* architecture shown in Figure 2.3. The model revolutionized NLP because of the *self-attention* mechanism, where RNN was removed entirely. The mechanism allows to make connections between different tokens and create contextualized embeddings. Moreover, Transformer uses the *multi-head attention* module that runs through the attention mechanism several times in parallel and then calculates an average of individual self-attention blocks [19]. It is useful for language understanding because a particular token can be important for multiple others. For example, the word *"going"* in the sentence: *"I am going to Prague tomorrow"* is related to *"Prague"* as well as *"tomorrow."* So one attention head might learn action-place and the other action-time relationship. This way, the model learns the context and language structure efficiently.

2.1.5 BERT

BERT [4] is a language model based on the encoder part of the Transformer architecture. It is designed to perform general language understanding tasks [4]. During the pre-training, it learns to understand language, grammar, and context, and during the fine-tuning, it is configured to adapt to a specific task. The model uses two strategies during the pre-training:

- *Masked Language Modeling (MLM)*. During the preprocessing, the model randomly chooses and masks words in a sentence, and then during training, it tries to output the

original sentence [20]. It makes BERT truly bidirectional, taking into account previous and following context simultaneously.

- The *Next Sentence Prediction (NSP)* technique is a classification layer of BERT, where during training, it takes two sentences and tries to predict if one goes after the other, helping BERT to better understand the relationship between pieces of text.

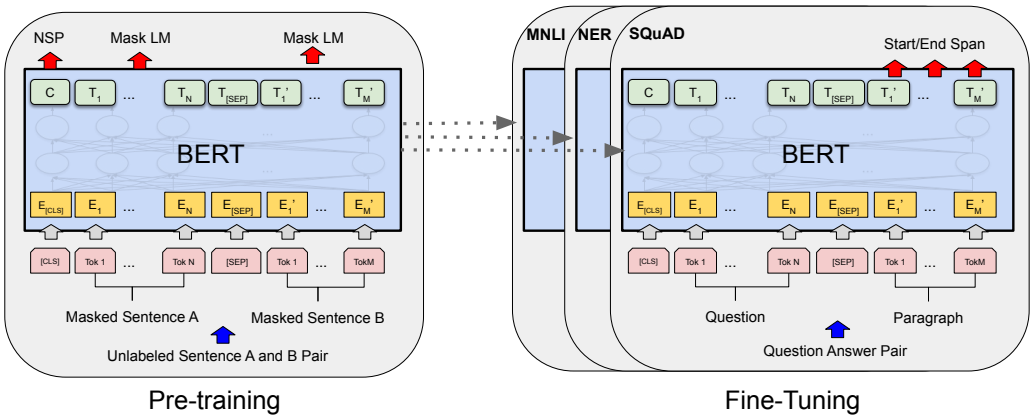


Figure 2.4 An illustration of BERT's pre-training and fine-tuning phases [4].

BERT uses the same architecture during pre-training and fine-tuning, but inputs and outputs are task-dependent. For example, instead of masked sentences, as shown in Figure 2.4, a Question Answering (QA) system inputs a question and a paragraph and asks a pre-trained model to output a text span from a paragraph containing the answer [4]. The fine-tuning usually just slightly modifies parameters learned during the pre-training.

The model's input is a tokenized text provided by the WordPiece tokenizer, which splits words into full forms or word pieces [21]. Individual tokens are transformed into vectors using the combination of *token*, *segment*, and *position* embeddings. Token embeddings are generated by WordPiece embeddings. Segment embeddings are vectors that tell BERT which part of a text contains a particular token. Position embedding represents the location of a token in a sentence [4].

BERT processes sentences in parallel, utilizing hardware more efficiently than RNN does. It makes training faster, so it is possible to train BERT on larger amounts of data, for example, on entire Wikipedia. At the time of publishing, BERT achieved state-of-the-art results on multiple tasks.

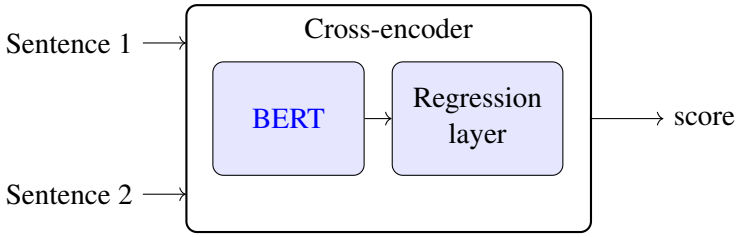


Figure 2.5 The cross-encoder architecture[2].

[2]The figure is adapted from https://www.sbert.net/docs/pretrained_cross-encoders.html

The model can be trained to output semantic similarity of sentences or documents by adding a regression layer on top of the encoder. Given two sentences as an input, it will return a real number representing their similarity. The described architecture is called a *cross-encoder*, see Figure 2.5. It achieves state-of-the-art performance on the STS task, but the implementation proved to be inefficient because, in some cases, embeddings are calculated multiple times [22].

2.1.6 Sentence Transformers

One of the main disadvantages of using BERT in sentence-pair regression tasks is that the cross-encoder architecture does computationally demanding operations without reusing previous calculations. For example, finding the most similar pair in a set of 10,000 sentences takes 49,995,000 inference computations or approximately 65 hours on a modern GPU [22]. So it is almost impossible to use classical BERT for applications that deal with semantic similarity.

Theoretically, it is possible to retrieve sentence embeddings by averaging token outputs provided by BERT, but the architecture is not built to generate sentence embeddings and would produce unsatisfactory results [22].

Researchers proposed the *bi-encoder* architecture [22], demonstrated in Figure 2.6, which is created to fine-tune BERT to output sentence embeddings. It aims to map all sentences to a vector space where embeddings of similar texts are close to each other. The architecture passes two sentences in a *siamese* fashion where input sentences are processed by two identical independent transformer-based sub-networks. The model uses a labeled dataset consisting of pairs of sentences with their respective similarity scores to fine-tune pre-trained BERT. It adjusts sentence embeddings so their similarity is close to the reference. The approach does not achieve the same performance on the STS task as a cross-encoder, but it produces better embeddings than those created by averaging BERT outputs [22]. More importantly, without a significant difference in performance, it offers the opportunity to precompute embeddings in advance.

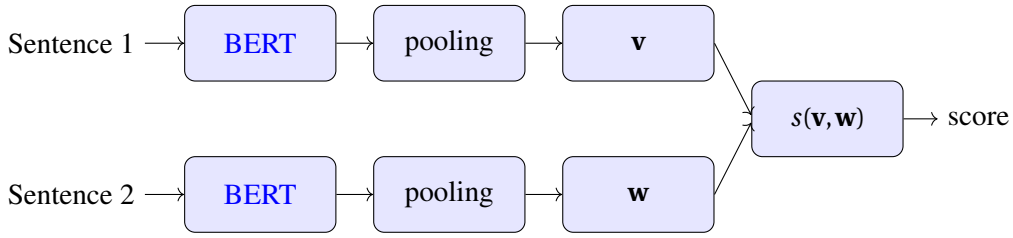


Figure 2.6 The bi-encoder architecture. In the pooling layer, the model computes the mean of all output vectors provided by BERT. The figure is adapted from [22].

A bi-encoder has advantages over a cross-encoder when used in real applications. Nevertheless, it requires additional fine-tuning in order to produce high-quality embeddings. Researchers in [22] fine-tuned and published several popular transformer-based models (including BERT) to output sentence embedding. They called the implementation *SentenceTransformers*. It is used throughout the rest of this work.

2.2 Experiments and results

2.2.1 Models

This section evaluates two transformer-based and one non-transformer model on the STS task. The models were selected based on their distinct architecture. The first is the BERT derivative called *a Robustly Optimized BERT pre-training Approach (ROBERTA)* [23]. The model does

not use the NSP technique because it was proved that it does not improve performance [23]. ROBERTA also made changes to MLM, such as performing masking during training, not the preprocessing. Researchers proved that BERT is significantly undertrained, so they trained ROBERTA on ten times more data than BERT, with the dataset size equal to 160GB of uncompressed text [23]. Sentence-Transformers implementation of ROBERTA is called Sentence-RoBERTa (S-ROBERTA). In this work, the pre-trained S-ROBERTA, called *all-roberta-large-v1*, is used. It was additionally fine-tuned on a dataset containing over a billion sentence pairs [24].

The second model is a transformer-based distilled language model called *MiniLM*. The distillation is a process of model compression, where a smaller model (student), in this case, MiniLM, is trained using outputs of a larger model (teacher). The model aims to mimic the self-attention mechanism of a transformer-based model, such as BERT or ROBERTA [25]. In this work, publicly available pre-trained MiniLM, called *all-MiniLM-L6-v2*, is used. It is distilled from BERT and is fine-tuned on almost the same data as *all-roberta-large-v1* (MiniLM is tuned on two additional datasets) [26]. MiniLM is generally faster and smaller but slightly less precise than BERT and its derivatives [25].

Pre-trained transformers are fine-tuned on two labeled datasets described in the following sections. They use the following hyperparameters during training:

- the batch size is 8, the maximum number of epochs is 10,
- the learning rate is $2 \cdot 10^{-5}$, the weight decay is 10^{-2}, and
- the optimization algorithm is Adam.

The third, non-transformer model Sent2Vec [8], provides static embeddings and is used as a baseline model for the performance comparison.

2.2.2 Metrics and evaluation

The performance on the STS task is evaluated using the following metrics:

- *Mean Absolute Error (MAE)* is an error metric that is calculated as follows:

$$\text{MAE} = \frac{1}{m} \sum_{i=0}^{m} \| y_i - z_i \|, \tag{6}$$

  where $\mathbf{y} = [y_1, \cdots y_m]$ and $\mathbf{z} = [z_1, \cdots z_m]$ are reference values and predicted similarity scores for m sentence pairs. Generally, the lower the MAE, the better the model fits the data.

- *Pearson correlation coefficient* indicates the linear relationship between two variables (in this case, vectors containing predicted values and reference scores). It is widely used in other works and also in this chapter to compare the performance of different models. Pearson coefficient r is calculated as follows:

$$r = \frac{\sum_{i=1}^{m} (y_i - \bar{y})(z_i - \bar{z})}{\sqrt{\sum_{i=1}^{m} (y_i - \bar{y})^2} \sqrt{\sum_{i=1}^{m} (z_i - \bar{z})^2}}, \tag{7}$$

  where $\bar{y}, \bar{z}$ are the mean values of $\mathbf{y}$ and $\mathbf{z}$. The larger values of the coefficient are better.

In the following sections, transformer-based models are trained on various datasets. The training is performed on NVIDIA GTX 1080Ti using CUDA toolkit[3], which allows software systems to process a code on a GPU. The inference is performed on an octa-core CPU (a part of the Apple M1 chip).

[3] https://developer.nvidia.com/cuda-toolkit

2.2.3 Data

The semantic textual similarity performance of the abovementioned models is evaluated on two datasets: STSB and "Harry Potter." The STSB dataset[4] is typically used for standard evaluation of different models which output semantic sentence similarities. It contains 5749 training, 1500 validation, and 1379 test samples. The validation set is used to evaluate a model during the fine-tuning and store the best-performing model. Each sample in the dataset contains a pair of sentences with a manually annotated score representing their similarity. The score is a number from zero to five, where zero means that sentences are not similar, while five represents their equivalence. During fine-tuning, the scores are normalized to scale between zero and one. Examples of labeled sentence pairs contained in the dataset are shown in Table 2.1.

First sentence	Second sentence	Score
A girl is playing on flute	A band is playing on a stage	0.26
A boy is playing an instrument	A man is playing a trumpet	0.50
Someone is slicing a vegetable	Somebody is slicing a tomato	0.60
Dogs swim in a pool	Dogs are swimming in a pool	0.84
A woman slices tofu	A woman is cutting tofu	1.00

Table 2.1 Example of pairs of sentences contained in the STSB dataset. The scores are normalized and rounded to two decimal places.

The STSB dataset contains commonly used sentences and phrases. However, some applications may work with unusual texts. For example, medical applications might contain some uncommon terminology that a model needs to learn in order to converse with doctors or nurses.

First sentence	Second sentence	Score
Wingardium Leviosa is taught to first-years at Hogwarts	Severus Snape teaches the potions class	0.00
Slytherin house is represented by a serpent	Harry Potter would fit well in Slytherin house	0.25
Capturing the Golden Snitch ends the game immediately	The Golden Snitch is worth 150 points	0.50
Harry Potter can be recognized by his cursed scar	Harry Potter has a lightning-shaped scar	0.75
Harry Potter can be recognized by his cursed scar	The Boy Who Lived can be recognized by his cursed scar	1.00

Table 2.2 Example of manually labeled pairs of sentences contained in the "Harry Potter" dataset.

For the purpose of the experiment, I created a small dataset containing sentences and facts from the book "Harry Potter" by J.K. Rowling. It consists of many unusual words and sentences, such as *"Avada Kedavra," "Wingardium Leviosa,"* or *"Hogwarts."* The dataset also uses

[4]Source of the dataset: https://ixa2.si.ehu.eus/stswiki/index.php/STSbenchmark

some common words in different meanings. For example, the word *"curse"* refers to an evil spell, and *"house"* is used as a synonym for the words "group" and "community."

The dataset has a similar structure to STSB. It contains 420 train, 106 validation, and 101 test pairs of sentences with manually assigned similarity scores (by the author). Any real value can be assigned to a pair of sentences, but for simplicity, a score assumes one of five values: $\{0.00, 0.25, 0.50, 0.75, 1.00\}$. Several sentence pairs with their respective similarity scores are shown in Table 2.2. The larger the score, the higher the similarity.

2.2.4 Comparison of embedding algorithms

Sentence-transformers allow fine-tuning a model using a labeled dataset, but since all BERT derivatives are trained on a large amount of data, it is not usually necessary. Table 2.3 shows that fine-tuning on the STSB training set improved the performance of both models by only 2-3%. The improvement is insignificant because the dataset uses common sentences that models understand well, so additional training does not bring much new information.

Embedding model	Fine-tuning	MAE	Pearson
sent2vec-twitter-unigram	None	0.20	0.76
all-MiniLM-L6-v2	None	0.15	0.83
	STSB training set	**0.13**	**0.86**
all-roberta-large-v1	None	0.14	0.84
	STSB training set	**0.11**	**0.88**

Table 2.3 Evaluation of Sent2Vec, MiniLM, and S-ROBERTA on the STSB test set.

Transformer-based models achieve good results on STSB. Table 2.3 also compares sentence transformers with the non-transformer-based Sent2Vec unigram model trained on the Twitter dataset [27]. Sent2Vec acquires 0.76 Pearson correlation coefficient, which is the best performance of all evaluated non-transformer-based pre-trained models [27]. It is clear that sentence transformers provide approximately 5-10% better performance than the best non-transformer model.

Embedding model	Fine-tuning	MAE	Pearson
sent2vec-twitter-unigram	None	0.31	0.85
all-MiniLM-L6-v2	None	0.20	0.91
	Harry Potter training set	**0.06**	**0.99**
all-roberta-large-v1	None	0.29	0.84
	Harry Potter training set	**0.05**	**0.99**

Table 2.4 Evaluation of Sent2Vec, MiniLM, and S-ROBERTA on the "Harry Potter" test set.

Transformer-based models are trained on a large amount of data. However, Table 2.4 shows that both models benefit from the fine-tuning in cases with uncommon terminology. Pre-trained

MiniLM performs approximately 10% better than pre-trained S-ROBERTA on the "Harry Potter" dataset, but the models achieve similar results after additional training. The fine-tuning improves the performance of MiniLM by 15% and of S-ROBERTA by 25%.

Sent2Vec cannot be fine-tuned in the same way as transformer-based models, it does not capture multiple meanings of words, and it performs worse on most of the datasets used in this work. For this reason, it will not be included in the following experiments regarding semantic textual similarity in this chapter.

2.2.5 Speed comparison

In AI-human conversation, the reaction time is crucial. It is essential to choose a model that has fast inference speed. The training and inference speeds of the models are shown in Table 2.5. Without much performance difference, MiniLM is significantly faster than S-ROBERTA, which makes it a good candidate for an embedding model in the intent detection module.

Embedding model	Training time [s/epoch]	Training speed [sent/s]	Inference speed [sent/s]
all-MiniLM-L6-v2	88	65	**1379**
all-roberta-large-v1	391	14	229

Table 2.5 Training and inference speed for all-MiniLM-L6-v1 and all-roberta-large-v1. Training was performed on the STSB dataset on NVIDIA GTX 1080Ti. Inference speed was measured on a CPU using the STSB test set.

2.2.6 Language understanding quality

At first glance, there is not much difference in performance between fine-tuned MiniLM and S-ROBERTA, as shown in Table 2.3 or Table 2.4. However, most sentences used in STSB and "Harry Potter" datasets are simple and do not measure how well models deal with more challenging examples.

First sentence	Second sentence	all-MiniLM-L6-v2	all-roberta-large-v1
Call me Tom	I will call you Tom	0.92	0.76
She is going to the park	She is going to park the car	0.80	0.53
The man took a bow to shoot	The man took a bow to the king	0.72	0.54
I went to a river bank	I went to a bank	0.80	0.71
She knows him better than I	She knows him better than me	0.98	0.98

Table 2.6 Comparison of semantic similarity scores of dissimilar pairs of sentences predicted by pre-trained MiniLM and S-ROBERTA models. Lower scores are better.

To evaluate language understanding quality, I included several sentence-pair examples that are tough for embedding models to distinguish. Table 2.6 shows that S-ROBERTA deals with more complicated examples better than MiniLM. Since MiniLM is a distilled model, it does not have a deep understanding of language. Nevertheless, the datasets used in this work are composed of primarily simpler sentences, where a distilled model might suffice.

2.2.7 Discussion

Transformer-based models, MiniLM and S-ROBERTA, perform reasonably well on the semantic sentence similarity task. Both models achieve 5-10% better results than the Sent2Vec unigram model trained on the Twitter dataset. However, the difference in performance between fine-tuned transformer-based models on both STSB and "Harry Potter" datasets is almost negligible (the difference is 1-2%). While providing good results, MiniLM completes training and inference phases several times faster than S-ROBERTA. MiniLM is a more suitable embedding model for the intent detection module, but S-ROBERTA has a better language understanding and can deal better with difficult examples.

I decided to use MiniLM as a default model in the intent detection module because of its speed and size. However, S-ROBERTA would probably be a more appropriate choice for different applications and other languages because it better understands the grammar and context. Both models are used to evaluate the module on the intent classification task, described in the following chapters.

Chapter 3

Intent classification

This chapter focuses on the intent recognition component. The first part addresses the main difficulties of the intent classification task and discusses strategies that can be applied to overcome them. The rest of the chapter explores two types of intent classifiers. The first is based on semantic textual similarity, which is used in many applications [28] [29]. It computes similarity scores between a user utterance and a set of templates and then labels the utterance with an intent to which the most similar sentence belongs.

The second approach treats intent classification as a standard text classification task. Modern state-of-the-art intent recognition systems often use BERT derivatives as their first layer and then place a simple classifier on top for predictions [30]. I decided to design a model's architecture that uses Sentence-transformers embeddings as its input. It presents the opportunity for precomputing sentence embeddings, making the algorithm efficient. This prospect is explored more in the following chapters, where the architecture of the intent detection module is discussed.

Both strategies are evaluated and analyzed in the last section of the chapter, which compares their accuracy and speed, keeping in mind usage in practical applications. Methods discussed in this chapter use numerical sentence representation as an input. These representations are provided by embedding models MiniLM and S-ROBERTA discussed in Chapter 2. Both of them are used and compared on the intent classification task.

3.1 Challenges of intent classification

The intent classification might seem like a straightforward task, but languages are complex, and any sentence can be expressed in many different ways. In order to build a reasonably performing classifier, several other difficulties must be considered.

3.1.1 Implicit intents

User intents can be generally divided into two groups: *explicit* and *implicit*. Explicit intents are precise and give clear requirements for a conversational assistant. These kinds of intents are easier for the system to understand. An example of an explicit intent can be: *"find affordable restaurants near me."* The message contains several keywords, and even a basic pattern-based intent detection system could work with this kind of text by detecting relevant keywords [31]. Implicit intents require additional analysis and speculation about what the user wants. For example, a user can ask: *"do I need to take an umbrella with me tomorrow?"* Here, a human would most likely understand that a person asks if it will rain tomorrow, and the target intent is the weather. However, it is challenging for AI because it must learn that "bringing an umbrella" is directly related to the weather. Therefore, implicit intents require additional work with sentence

semantics, and algorithms must use high-quality sentence embeddings to perform adequately in such cases.

3.1.2 Typographical errors and misspelled words

Typographical and grammatical errors are very common in written language [32]. Typographical errors are often relatively easy to correct, but sometimes they can change the meaning of the whole sentence. For example, modifying just one letter can change *"The application is now available on the iPad"* to *"The application is not available on the iPad."* While hardest to correct, these examples are relatively rare. Often, errors do not change the meaning, but their presence can make the classification very difficult. It was found that embedding models do not work well with texts containing errors [33].

In order to improve a system's performance when classifying messages containing errors, two approaches can be used:

- **Spell correction.** An additional, often machine learning-based, preprocessing step can improve the performance but slow down the inference phase. This approach is used in the final intent detection module.

- **Data augmentation.** Performance improves when training data are augmented with sentences containing errors [34]. This step helps when dealing with omitted articles or other grammatical errors, and it usually does not slow down the inference phase in contrast to spell correction.

3.1.3 Coverage limitations

The ability to classify intents is often limited to a predefined set that AI can understand. Limitations cause a system to misclassify users' intents that are not present in a dataset. In this case, a conversational assistant might fail to fulfill its purpose of use [35]. Acquiring and processing large amounts of data is expensive, so most systems' prediction capability is limited. Building a system with a pre-trained classifier able to understand any topic is not the purpose of this work. However, the intent detection module is built to allow users to add custom intents, which are then used for training and classification.

3.2 Intent classification approaches

The first type of intent classifier is based on semantic sentence similarity. The method will be further referred to as *unsupervised* because no additional data are used for training. The second approach is a neural network-based classifier. It takes a sentence embedding as an input and predicts the intent to which a message belongs. This method will be referred to as *supervised* because it needs additional learning to function correctly.

3.2.1 Unsupervised approach

The unsupervised intent classifier is a direct application of semantic sentence similarity discussed in Chapter 2. It uses the fact that two similar sentences have similarly oriented embeddings. The approach aims to find a sentence in a predefined set that is the most similar to an input query.

In an actual application, all predefined variations are transformed into vectors during the initialization. An encoder creates a set of templates $T = \{\mathbf{t}_1, \mathbf{t}_2 \dots \mathbf{t}_m\}$, where m is the number of

samples (sentences) in a dataset, and stores it for repeated usage. This approach is time-efficient because embeddings need not be calculated repeatedly.

The method starts the classification of a user utterance by creating its vector representation $\mathbf{v}$. Then, it compares it with all the template embeddings using the cosine similarity score. The closest embedding $\mathbf{t}^*$ is found using the formula:

$$\mathbf{t}^* = \underset{\mathbf{t} \in T}{\operatorname{argmax}}\ s(\mathbf{t}, \mathbf{v}) = \underset{\mathbf{t} \in T}{\operatorname{argmax}} \frac{\mathbf{t} \cdot \mathbf{v}}{\|\mathbf{t}\| \|\mathbf{v}\|}. \tag{8}$$

According to the algorithm, $\mathbf{t}^*$ and $\mathbf{v}$ belong to the same intent, so it classifies a given query with a label equal to the label of $\mathbf{t}^*$. The approach is illustrated in Figure 3.1.

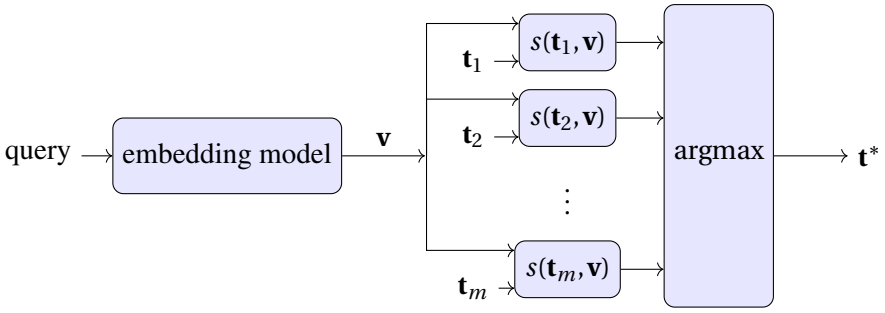


Figure 3.1 Illustration of the semantic similarity-based intent classifier. Variables *query* and $\mathbf{v}$ represent a user utterance and its respective embedding. Vectors $\mathbf{t}_1, \mathbf{t}_2, \mathbf{t}_m$ are precomputed representations of sentences from a set of m templates.

The unsupervised approach has several advantages:

- It does not require additional training.
- The results can be easily interpreted because of the semantic similarity score. They can also be used for future analysis.
- The model does not need to be updated to work with new data when templates are changed.
- It is easier to serve because the embedding model is the only machine learning component present.

One of the method's main disadvantages is that it can be slow when the number of templates is large because it needs to compare a query with each template.

3.2.2 Supervised approach

The supervised approach uses a two-layer feed-forward neural network shown in Figure 3.2 and is designed as follows:

- The input layer consists of n neurons. The value on n is equal to the dimensionality of sentence embedding, which depends on the encoder. For example, MiniLM produces 384-dimensional embeddings, while S-ROBERTA uses 1024-dimensional.
- The output layer consists of k neurons, where k is the number of classes (intents). For example, $k = 150$ when a dataset with 150 intents is used.

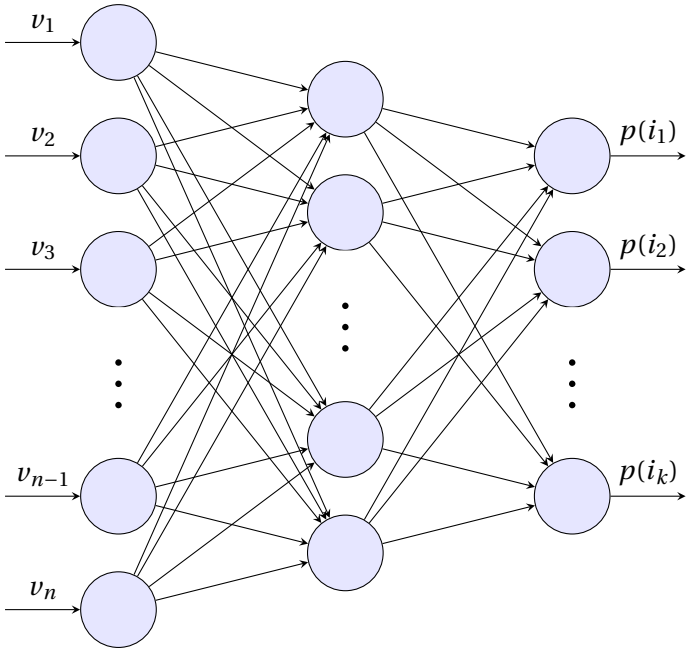


Figure 3.2 A two-layer feed-forward neural network for intent classification.

- The number of neurons in the hidden layer was determined empirically using the validation part of the data. I have tried 100, 150, 200, 250, and 300 neurons, but the model performs best with 200 neurons in the hidden layer.

During training, the model uses the following parameters:

- the activation function is Rectified Linear Unit (ReLU),
- the learning rate is 10^{-3}, the weight decay is 10^{-6}, the dropout probability is 0.25,
- the batch size is 64, the number of iterations is 5000, and the maximum number of epochs is 20.

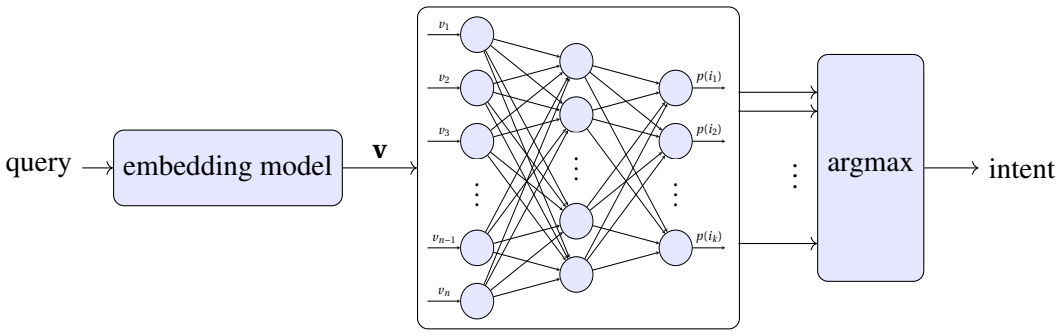


Figure 3.3 Illustration of the neural-network-based intent classifier. Variables *query* and $\mathbf{v}$ represent a user utterance and its respective embedding. The vector $\mathbf{v} = \begin{bmatrix} v_1 & v_2 & \dots & v_n \end{bmatrix}^\top$ is passed to a neural network that inputs its individual elements.

Figure 3.3 illustrates the architecture of described intent classifier. The proposed method is more computationally efficient than the unsupervised algorithm because it does not have to process each template during the inference phase. The neural network-based intent classifier has other advantages over the unsupervised approach:

- A model uses intent variations only during training. Therefore, it does not need to store and process them during the inference.
- The approach is faster because it does not need to process the whole template set.
- It can learn other patterns in embeddings belonging to the same intent, not just the similarity.

The main disadvantage of the approach is that it needs additional training. However, it uses a relatively uncomplicated model, which can be trained in several seconds.

3.2.3 Intent classification with BERT

As was briefly mentioned at the beginning of this chapter, many state-of-the-art intent classifiers use classical BERT and its derivatives for classification tasks. Therefore, I evaluate BERT and ROBERTA in the following experiments, where this approach is referred to as *BERT-based*. Figure 3.4 shows the resulting classifier, which has a similar structure to the supervised approach. However, in contrast to the supervised approach, embeddings cannot be precomputed, so the training takes a long time (30 minutes per epoch on a CPU).

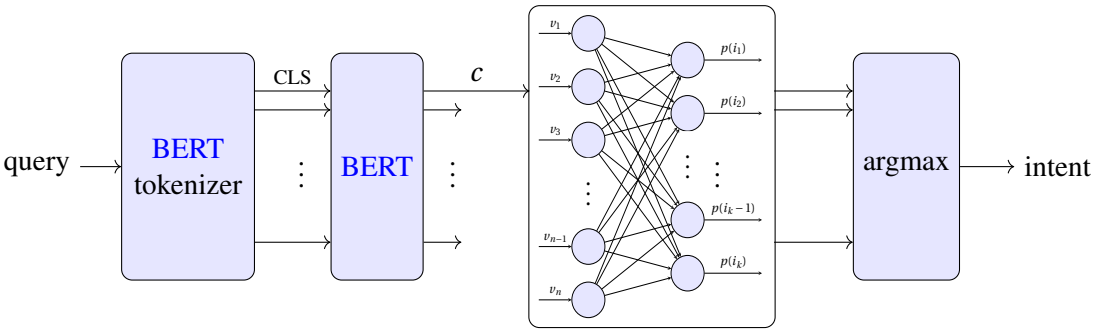


Figure 3.4 Illustration of the BERT-based intent classifier. An encoder processes all tokens of the variable *query*, but only the CLS token's encoding c is used as a neural network input.

BERT inputs a sequence of tokens provided by the WordPiece tokenizer. Each sequence starts with a special classification token CLS, which represents the encoding of the whole sentence, and is usually used as an input to a text classifier [4]. The same classifier (logistic regression) and the same hyperparameters are used as described in [36]:

- the maximum number of input tokens is 50,
- the learning rate is $5 \cdot 10^{-5}$, the dropout probability is 0.25,
- the batch size is 128, and the maximum number of epochs is 20.

ROBERTA is very similar to BERT, but instead of WordPiece, it uses the Byte-Pair Encoding (BPE) tokenizer and does not perform NSP. During the fine-tuning, it uses the same architecture and hyperparameters as BERT.

3.3 Experiments and results

3.3.1 Data

The intent classifiers are compared on the two datasets. The SNIPS dataset is widely used as a benchmark for the intent classification and slot filling tasks [37]. The second dataset, unofficially called CLINC150, consists of more realistic data and a larger variety of intents [38].

SNIPS[1] consists of 13084 training, 700 validation, and 700 test samples distributed between seven intents, as shown in Table 3.1.

Label	Intent	Number of training samples	Number of validation samples	Number of test samples
0	PlayMusic	1914	100	86
1	AddToPlaylist	1818	100	124
2	RateBook	1876	100	80
3	SearchScreeningEvent	1852	100	107
4	BookRestaurant	1881	100	92
5	GetWeather	1896	100	104
6	SearchCreativeWork	1847	100	107

Table 3.1 Numbers of training, validation, and test samples in each class.

Several examples of sentences found in the dataset are shown in Table 3.2. Some intent pairs, like *PlayMusic* and *AddToPlaylist*, share entities, for example, a band or music genre. Others, like *SearchCreativeWork*, sometimes use the same topic as *SearchScreeningEvent*. These pairs are more problematic for models to distinguish and sometimes might be mixed up.

Label	Intent	Example
0	PlayMusic	Play the top-20 best Chicane songs on Deezer
1	AddToPlaylist	Add this track to my global Funk
2	RateBook	Rate this series a 5
3	SearchScreeningEvent	Find Fish Story
4	BookRestaurant	Book a restaurant at sixteen o clock in SC
5	GetWeather	Will it be colder in Ohio
6	SearchCreativeWork	Show me the picture Creatures of Light and Darkness

Table 3.2 Examples of the sentence contained in the SNIPS dataset.

[1] https://github.com/sz128/slot_filling_and_intent_detection_of_SLU/tree/master/data/snips

The CLINC150[2] dataset consists of *in-scope* and *out-of-scope* examples divided into training, validation, and test sets, as shown in Table 3.3.

Dataset	Number of training samples	Number of validation samples	Number of test samples
In-scope	15000	3000	4500
Out-of-scope	100	100	1000

Table 3.3 Number of training, validation, and test samples contained in each part of CLINC150.

Each in-scope sample is a sentence belonging to one of the 150 intents. The intents include the meaning of life, improving credit score, fun facts, booking a hotel, and many others. Each of them consists of precisely 100 training, 20 validation, and 30 test variations. The out-of-scope samples are sentences that do not belong to in-scope intents supported by the system [38]. Examples of in-scope and out-of-scope intents contained in the dataset are shown in Table 3.4.

Label	Intent	Sentence variation
0	translate	How do i say 'hotel' in Finnish
		What is the equivalent of 'life is good' in French
4	meaning_of_life	What is the real meaning of life
		Why do you think we're here
94	order_status	What is the status of my order
		Can you update me on my last Amazon orders
149	card_declined	Why did my card not get accepted then
		Why was my card not accepted yesterday
150	out-of-scope	How can i become an aerospace engineer
		Have you seen the remote

Table 3.4 Illustration of sentence variation and intents contained in the CLINC150 dataset. Examples were purposely capitalized by the author for better visualization.

Data contained in CLINC150 are very realistic because users often use conversational systems for tasks that they are unable to perform. For example, a user can ask about the weather in a cooking application that does not contain any intents regarding the weather. CLINC150 includes many such examples and, therefore, is suitable for estimating performance and quality in real applications.

[2]Source of the dataset: https://github.com/clinc/oos-eval

3.3.2 Metrics and evaluation

In this section, the following metrics and evaluation approaches are used:

- **Accuracy** is the primary metric used for the evaluation. It is calculated as the number of correctly classified samples over the size of a dataset.
- **Confusion matrix**. Rows of the matrix represent the reference class, while columns illustrate the predicted class. Each element contains the count values for each pair of actual and predicted classes. The diagonal of the matrix counts the correctly predicted values, while non-zero elements outside of the diagonal contain error counts. The ideal confusion matrix is diagonal, having non-zero values only on the main diagonal.
- **K-fold cross-validation** is used for the evaluation on a small dataset. The technique divides data into k groups. One group acts like a test set and the rest as a training set. After evaluation, the next portion of data becomes a test set. The procedure is repeated k times. The resulting accuracy is equal to an average of k estimates.

Training and evaluation are performed using the same hardware as described in Chapter 2.

3.3.3 Evaluation on the SNIPS dataset

Three abovementioned intent classifiers are evaluated on the test part of the SNIPS dataset. Table 3.5 shows that all methods perform relatively similarly, but the supervised approach is slightly better than the unsupervised when the MiniLM embedding model is used. BERT-based methods perform approximately 1% better than S-ROBERTA-based classifiers. Note that the performance of BERT on the SNIPS dataset is lower by 1% than the one published in [36] (even though the same hyperparameters and approach were used).

To this day, the best performance on the dataset was achieved by the dual intent-entity model introduced in [39]. The duality of the model influences the performance of the intent classifier, which achieves 99% accuracy on the test set.

Method	Unsupervised		Supervised		BERT-based	
Encoder	all-MiniLM L6-v2	all-roberta large-v1	all-MiniLM L6-v2	all-roberta large-v1	BERT	ROBERTA
Accuracy	0.92	**0.96**	0.95	**0.96**	**0.97**	**0.97**

Table 3.5 The accuracy of the unsupervised, supervised, and BERT-based classifiers on the SNIPS test set.

The classified intents are visible in the confusion matrix in Figure 3.5. It shows that all approaches work similarly, although the unsupervised misclassifies *SearchCreativeWork* intent more often than other approaches. It is not surprising considering that this intent often contains sentences that share a topic with other intents. For example, *SearchCreativeWork* and *SearchScreeningEvent* both include sentences about movies and shows, see Table 3.6 for examples. Confusion matrices for the unsupervised and supervised S-ROBERTA-based methods are not present because they are similar to MiniLM-based.

The dataset will not be explored in further experiments because, like most other popular benchmark datasets, it has a relatively small number of intents and a very high number of training samples per intent which is hard to obtain. Moreover, the number of intents in real applications is often much higher. So the following experiments focus on another dataset that is close to real-world data.

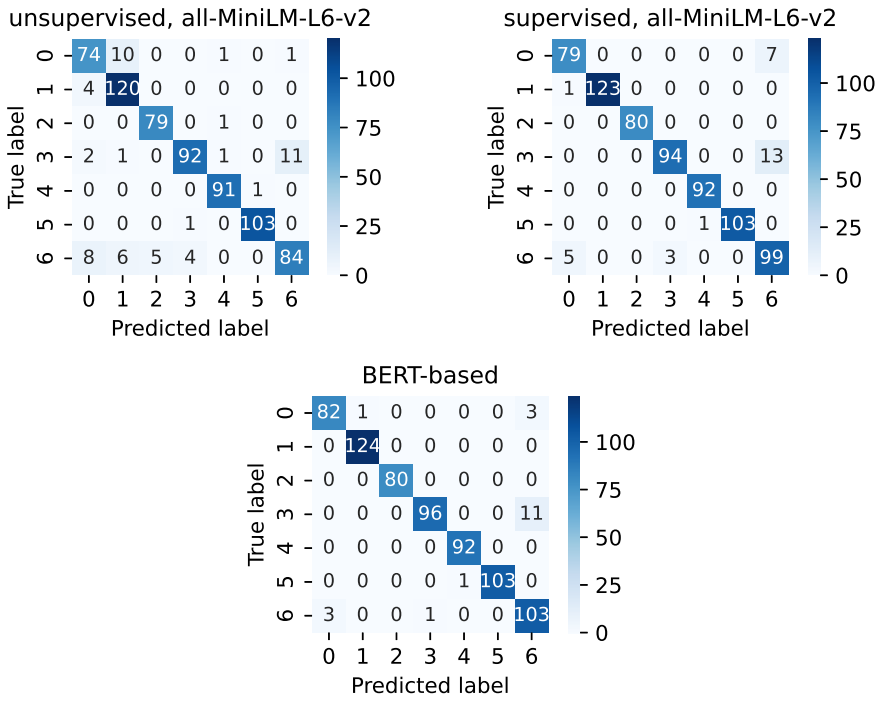


Figure 3.5 Confusion matrices for different intent classification approaches evaluated on the SNIPS dataset. For better visualization, the categories labels are used instead of names: 0=*PlayMusic*, 1=*AddToPlaylist*, 2=*RateBook*, 3=*SearchScreeningEvent*, 4=*BookRestaurant*, 5=*GetWeather*, 6=*SearchCreativeWork*.

Sentence	Matched sentence	Predicted intent	True intent
Find a movie called Living in America	Find out what films are playing at North American cinemas	SearchScreeningEvent	SearchCreativeWork
In one hour find King of Hearts	Search for To Heart 2	SearchCreativeWork	SearchScreeningEvent

Table 3.6 Examples of sentences that are misclassified by the MiniLM-based unsupervised approach.

3.3.4 Evaluation on the CLINC150 dataset

The performance of unsupervised, supervised, and BERT-based approaches is compared using the test part of the CLINC150 dataset. The validation data are used for evaluating a model during the training. The approaches are first evaluated on the dataset's in-scope part, omitting any other examples. Then the in-scope training set is combined with the out-of-scope training set to evaluate a method on out-of-scope test samples. Table 3.7 shows that the supervised approach achieves approximately 4-8% better accuracy on the in-scope and 20-30% better accuracy on the out-of-scope set than the unsupervised.

Neither model reacts well to sentences that do not belong to supported intents. Even the best accuracy, which is 50% achieved by the ROBERTA-based classifier, is unsatisfactory. So in the following sections, a method for handling the out-of-scope intents will be explored.

The performance of BERT-based classifiers on SNIPS and the in-scope part of CLINC150 is very similar to the supervised approach. However, these models cannot precompute encodings

and are much harder to train. One epoch takes approximately 30 minutes on a CPU, and the model needs several epochs to perform well. It is expected that the content of an application will be updated on a regular basis. In this case, the BERT-based method is not a practical approach, so I decided not to explore it further. In the rest of the work, the supervised approach refers to sentence transformers with a supervised layer described in subsection 3.2.2. It has similar architecture and properties as BERT-based classifiers, but it is much more efficient because of the opportunity to precompute sentence embeddings. Moreover, in contrast to BERT, the encoding results are meaningful and can be used for further analysis.

Method	Unsupervised		Supervised		BERT-based	
Encoder	all-MiniLM L6-v2	all-roberta large-v1	all-MiniLM L6-v2	all-roberta large-v1	BERT	ROBERTA
IS accuracy	0.88	0.92	0.94	**0.96**	**0.96**	0.95
OOS accuracy	0.10	0.11	0.32	0.43	0.43	0.50

Table 3.7 The evaluation of described methods on the CLINC150 dataset. (IS refers to in-scope and OOS refers to out-of-scope data).

The overhaul accuracy of the intent classifier on the in-scope set is relatively high. However, it is worth knowing which examples the models misclassify and why. In order to investigate it, I studied the confusion matrix (which is hard to visualize) of the supervised approach based on MiniLM embeddings and found that the classes with the most incorrect predictions are: *change_user_name*, *shopping_list*, and *change_ai_name*. These intents have 70% precision.

Sentence	**Predicted intent**	**Actual intent**
I'll pass	calculator	no
What's your name by the way	user_name	what_is_your_name
Is Joey my name	change_user_name	user_name
Can I start calling you Dave	change_user_name	change_ai_name
Will you start calling me Chaz	change_ai_name	change_user_name
How can I say "cancel my order" in French	cancel_reservation	translate
Define monetary for me please	exchange_rate	definition
You are not wrong about that	no	yes

Table 3.8 Examples of misclassified sentences with their respective predicted and true intents. The predictions are generated by the supervised approach that uses MiniLM embeddings. Examples were purposely capitalized by the author for better visualization.

Table 3.8 shows that the intent *change_user_name* is sometimes incorrectly classified as *change_ai_name* and the other way around. The same happens with intents *change_ai_name* and *user_name*. The table also shows that sometimes typical variations of one intent are contained in another. For example, *"cancel my order"* refers to the cancellation, while *"how to say*

'cancel my order' in French" belongs to the *translate* intent. I also studied the confusion matrix and sentences that were incorrectly classified by the S-ROBERTA-based classifier, and while it contains some incorrectly labeled samples, it usually confuses sentences that are very similar in meaning.

Mentioned intents are problematic for both, the unsupervised and the supervised approaches. However, other kinds of sentences that are typically misclassified by the unsupervised approach will be studied further in later sections.

3.3.5 Improvement of the out-of-scope accuracy

The accuracy on the out-of-scope dataset is relatively low because any possible sentence which the model does not support is considered out-of-scope. It is hard to train a model on such data because they do not have a specific pattern. A possible way to overcome this problem is by using a threshold. If a model is unsure, it can say that a sentence is out-of-scope. In other words, if a model, which is trained on the in-scope data, returns confidence that is less than a threshold value, an utterance is labeled as out-of-scope. The results for different threshold values are shown in Figure 3.6.

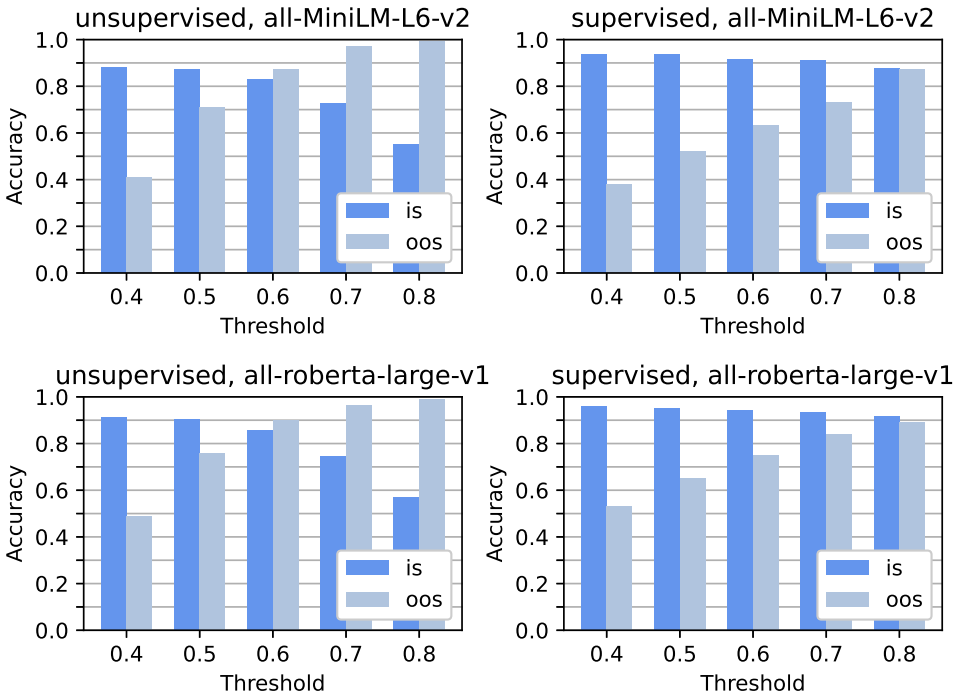


Figure 3.6 Evaluation of models on the in-scope and out-of-scope validation datasets. A model is trained on the in-scope training set and evaluated on the in-scope and out-of-scope validation set. When a model's confidence is less than a threshold value, an input sample is classified as out-of-scope.

The usage of a threshold decreases the performance of a model on the in-scope set. However, Figure 3.6 shows that the supervised approach is more resistant to it. It happens because the method usually returns a high score (around 0.90-0.95) when it classifies a sample correctly and a relatively low score (around 0.65-0.70) when it misclassifies it. The accuracy of the unsupervised approach on the in-scope set is more affected by choice of a threshold, so it is not as flexible as the supervised method.

The choice of a threshold depends on the application. In the case of conversational assistants, where users can write out-of-domain messages, a suitable threshold value can be around 0.8 for

the supervised approach and 0.6 for the unsupervised. It gives approximately 90% (supervised) and 85% (unsupervised) in-scope and 80% (supervised) and 85% (unsupervised) out-of-scope accuracy on the test sets.

3.3.6 Impact of the dataset size on the performance

CLINC150 has a much smaller number of variations per intent than SNIPS, but it still contains 100 variations available for training. It is sometimes difficult and expensive to collect a training set of this size in real applications, especially when the number of intents is high. Therefore, I decided to test how the performance changes depending on the number of variations.

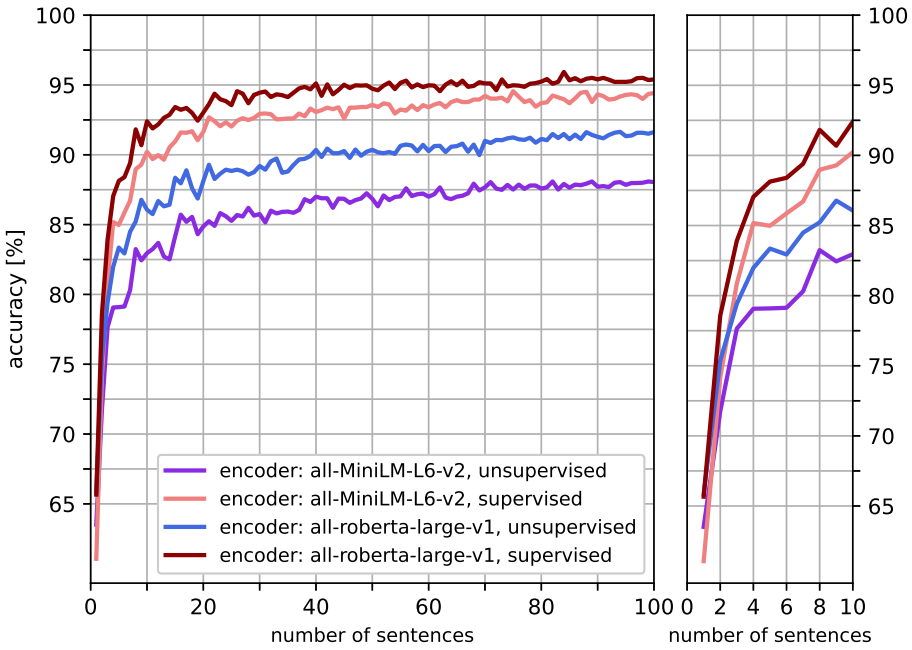


Figure 3.7 An average prediction accuracy of the unsupervised and supervised methods on the CLINC150 (in-scope) dataset depending on the number of sentences per intent. The graph on the right displays the same data as the one on the left but for a smaller number of sentences for better visualization.

The evaluation is performed on the in-scope test set described in subsection 3.3.4. The training set consists of intents having from one to 100 variations. Depending on the approach, the number of variations impacts the following:

- For the unsupervised approach, it limits the number of sentences in the template set T to which an algorithm compares a query.
- For the supervised approach, it limits the training dataset's size.

Figure 3.7 demonstrates that only three variations are needed to achieve accuracy above 75%. The unsupervised approach achieves 78% (MiniLM) and 79% (S-ROBERTA) accuracies on the dataset having three variations per intent, while the supervised achieves 81% (MiniLM) and 84% (S-ROBERTA). It is worth mentioning that the accuracy of the unsupervised approach is continuously increasing with an increasing number of variations. However, after approximately 20 variations, the change in performance is negligible (only 0.5% for every 20 added

variations). On the other hand, the supervised approach's performance converges at around 30 training samples and barely changes with additional samples. To summarize the results:

- At least three variations are needed for adequate performance.
- 20 is a suitable number of variations for an optimal cost-performance ratio.
- Having more than 30 variations is impractical and provides minimal accuracy improvement.

Figure 3.7 also confirms that the supervised approach performs better than the unsupervised. It can be explained by the fact that the neural network-based classifier uses more features than just semantic textual similarity. It can be a notable advantage when working with implicit intents because they are sometimes distinct from predefined variations.

3.3.7 Classification of queries containing entities

Most user messages contain some entities such as places, names, and languages. Entities can substitute one another, creating another variation of the same intent. For example: *"add bread to the grocery list"* contains the entity *"bread."* However, any food product or drink can replace *"bread,"* and the sentence's topic stays the same. A training set has a limited amount of entity variations. However, any entity values are possible in an actual application, and it is hard to maintain a list of all the possibilities. Therefore, it is worth knowing how intent classifiers react to sentences containing entities that were not present in the training set.

Sentence	Matched sentence	Predicted intent	True intent
Is chocolate cake healthy to eat	How many calories are in chocolate cake	calories	nutrition_info
Is broccoli healthy to eat	What's the best way to make a broccoli salad	recipe	nutrition_info
I am Terry	Share my location with Brad and Terry	share_location	change_user_name
I am Natalie	Please send a text message to natalie for me	text	change_user_name
Show me something funny about reptiles	Have any cool facts on reptiles	fun_fact	tell_joke
Show me something funny about flamingos	What's an interesting fact about flamingos	fun_fact	tell_joke

Table 3.9 Pairs of sentences that are considered matching by the MiniLM-unsupervised approach. For these examples the MiniLM-supervised classifier provided accurate predictions.

Table 3.9 shows that the unsupervised approach often misclassifies examples when they contain entities present in training sentences that belong to other intents. The supervised approach deals with this scenario better and classifies all sentences correctly.

3.3.8 Speed comparison

A typical application considered in this work uses a relatively small number of intents. In some applications, however, there can be significantly more of them. The unsupervised approach can precompute the embeddings, but it still needs to compare a query to every template at run-time. Therefore, it could take a long time when the list of intents is high.

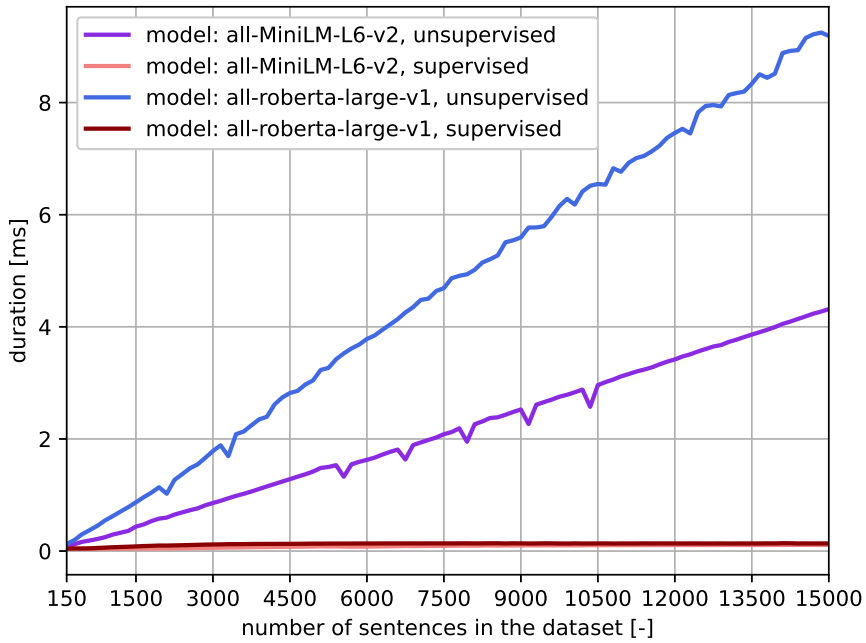


Figure 3.8 Dependency of one-sample classification time on the number of intent variations. The measurement includes only the classification phase, a computational time of query encoding is not present. The resulting duration is the average of five measurements. The classifier was trained and tested on the CLINC150 in-scope dataset.

Figure 3.8 shows the dependency of classification time on the number of variations. With a growing number of variations, the unsupervised method's inference time grows linearly, and the supervised classification duration is constant. It is worth mentioning that even though the embedding generation is not included in the measurement, the computation time of an algorithm is higher when it uses S-ROBERTA because of the embedding sizes. S-ROBERTA produces 1024-dimensional vectors while MiniLM generates 384-dimensional.

The slowest part of the inference phase is usually the creation of an input query embedding (around 8 ms for MiniLM, including the transformation of a vector into a tensor). However, in contrast to classification, the encoding speed does not depend on the number of training examples and would just add some constant time to each measurement shown in Figure 3.8.

3.3.9 Usage in the target platform

The target company aims to integrate the intent detection module as a new solution for its intent recognition system. It currently uses an enriched version of an intent classifier based on semantic sentence similarity of embeddings provided by the Sent2Vec algorithm. It will be referred to as *modified Sent2Vec*. For completeness, I will also include a supervised version

of modified Sent2Vec, called *supervised Sent2Vec*, which uses the same neural network-based intent classifier as described in subsection 3.2.2.

The solutions are compared on SNIPS, CLINC150, and internal company data (appointment booking related). The internal dataset is relatively small and does not has a capacity for train/test split, so modified Sent2Vec and proposed intent classifiers will be evaluated using the k-fold cross-validation.

		SNIPS	**CLINC150 in-scope**	**CLINC150 out-of-scope**
Modified Sent2Vec		0.94	0.84	0.08
Unsupervised	all-MiniLM-L6-v2	0.92	0.88	0.10
	all-roberta-large-v1	**0.96**	0.92	0.11
Supervised Sent2Vec		**0.96**	0.92	0.48
Supervised	all-MiniLM-L6-v2	**0.95**	0.94	0.32
	all-roberta-large-v1	**0.96**	**0.96**	0.43

Table 3.10 The evaluation of the current solution used in the company and its supervised version on the SNIPS and CLINC150 datasets. The results of MiniLM and S-ROBERTA-based methods are shown for comparison.

Table 3.10 shows that all mentioned approaches perform similarly on SNIPS, and the supervised methods based on MiniLM and S-ROBERTA outperform Sent2Vec on the in-scope dataset. Neither evaluated method was able to learn to recognize out-of-scope sentences adequately.

Fold	**all-MiniLM-L6-v2**		**all-roberta-large-v1**		**Modified Sent2Vec**	**Supervised Sent2Vec**
	unsupervised	supervised	unsupervised	supervised		
1	0.83	0.95	0.83	0.92	0.70	0.78
2	0.73	0.85	0.78	0.90	0.67	0.85
3	0.73	0.82	0.73	0.83	0.67	0.83
4	0.73	0.83	0.78	0.88	0.67	0.72
avg.	0.76	**0.86**	0.78	**0.88**	0.70	0.80

Table 3.11 Comparison of the designed intent classifiers with the current intent classifier used in the company on real-world data.

Table 3.11 shows that both intent classification approaches designed in this work achieve better accuracy on internal data than the current solution used in the organization. It also confirms that transformer-based approaches outperform Sent2Vec-based methods with the same architecture on most of the datasets used in this work (except SNIPS, where they have similar performance). These results are not surprising, considering that pre-trained MiniLM and S-ROBERTA perform considerably better on the STSB dataset than pre-trained Sent2Vec (0.83, 0.84, and 0.76 Pearson correlation coefficient, respectively). The S-ROBERTA-based approach achieves approximately 2% better results than the MiniLM-based in this evaluation.

3.3.10 Discussion

All the abovementioned approaches have their advantages and disadvantages. Nevertheless, the transformer-based supervised approach is the most suitable for the intent classification task. It is more accurate, works better with sentences containing entities, and is relatively fast even when the training set is large. The supervised approach is also memory efficient because it does not have to store templates and embeddings. Because of these reasons, the neural network-based classifier is used as a part of the intent detection module.

Chapter 4

Machine learning life cycle

The intent detection module consists of two machine learning components: the embedding model and the neural network-based intent classifier. The development of a machine learning component incorporates data preparation, model development, and model deployment [40]. These linked phases form a cyclical process called *the machine learning life cycle*, shown in Figure 4.1. The cycle illustrates consistent updates of machine learning models. Every dataset modification requires the revision of the model's architecture, which is typically followed by the deployment of a new model. After some time, updated data arrive, and the cycle repeats itself.

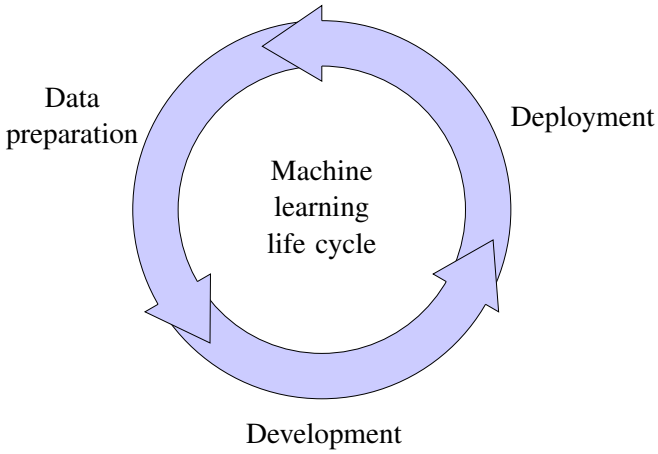


Figure 4.1 Three steps of a machine learning life cycle.

4.1 Data preparation

Data preparation is a crucial step in any machine learning project. The dataset's quality is often more critical than the sophistication and superiority of a model.

4.1.1 Data acquisition and labeling

Each task works with different data and requires a specific dataset. For example, the embedding model adjusts vector representations using a set of sentence pairs and their cosine similarity scores. The intent detection module allows users to use their own data to tune the embedding model. An example of model fine-tuning on the "Harry Potter" dataset is described in subsection 2.2.3. The dataset consists of various sentences with uncommon terminology and

expressions. These sentences are combined into pairs and manually labeled. In this case, there are only five possible labels (0.00, 0.25, 0.50, 0.75, and 1.00), but it is not a general rule, and in other datasets, labels can assume any real number.

Intent detection is a classification task that uses a dataset consisting of predefined intents. Each of them has a certain number of variations that are not just similar sentences and paraphrases but general sentences belonging to a topic. For example, CLINC150 contains the following sentences:

- *"In Austin, what are some tourist places to go to."*
- *"Find something fun for me to do in Dallas."*

The sentences above are not similar, but they belong to the same intent, called *travel_suggestion*. A set of intent variations might contain paraphrases and similar sentences, but, most importantly, it should sufficiently cover each intent's variations.

Datasets are rarely unchangeable. Intents and variations might be added or removed, and there is no guarantee that a model will perform well on another set of data. Therefore the whole machine learning life cycle should be reviewed with every update.

4.1.2 Data cleaning

Machine learning models perform well when presented with the data they expect. A dataset containing errors or incorrect labels might disturb the learning process and cause a decrease in performance. Hence, it is often required to clean data before the training.

Datasets for tuning the embedding model and training the intent classifier need similar processing steps. They include deleting duplicates, adding missing values, and removing invalid and corrupted data. As a final step, the spellchecker processes the resulting sentences. The intent detection module uses a fast and simple spellchecker called SymSpell[1]. It processes each word in a sentence, and when it encounters an unknown word, it edits each character until a modified word is present in the vocabulary. If there are multiple possibilities, the most frequently used word replaces the original.

SymSpell and similar spellcheckers, such as LinSpell[2] or TextBlob[3], can correct typographical errors but cannot fix grammatical errors. More sophisticated spellcheckers, such as Neuspell[4] or contextualizedSpellCheck[5], are machine learning-based models. They analyze the context, suggesting a more appropriate word when multiple possible modifications are present. Machine learning-based spellcheckers do not always work well. Sometimes they change a sentence entirely. For example, contextualized spellcheck changes the sentence *"this is riidiculous"* to *"this is because"* while SymSpell deletes unnecessary "i" and produces the correct sentence, which is *"this is ridiculous."* There are better machine learning-based spellcheckers, such as Bing Spell Check[6], but they are not free, so they are not used in the intent detection module.

4.1.3 Data augmentation

Generating additional data helps increase the training dataset size and improve a model's performance. The intent detection module augments data using the *synonym replacement* technique,

[1] https://github.com/wolfgarbe/SymSpell
[2] https://github.com/wolfgarbe/LinSpell
[3] https://github.com/sloria/TextBlob
[4] https://github.com/neuspell/neuspell
[5] https://github.com/R1j1t/contextualSpellCheck
[6] https://www.microsoft.com/en-us/bing/apis/bing-spell-check-api

which performs the substitution of words with their synonyms [41]. For example, the "Harry Potter" dataset contains many uncommon terms. The embedding model learns to understand the language better by replacing these terms with synonyms. It creates a new training sample with sentences equivalent to the original with all entities substituted with their synonyms. For example, the original "Harry Potter" dataset contains the sample:

- *"The Wingardium leviosa spell is taught to first-years at Hogwarts,"*

and the augmented "Harry Potter" set contains the following additional sentence:

- *"The Levitation Charm is taught to first-years at Hogwarts."*

This way, the embedding model learns that *"Wingardium Leviosa"* and the *"Levitation Charm"* are equivalent.

Other data augmentation techniques include random word insertion, random words swap, random deletion, lemmatization, and many others [42]. The intent detection module does not implement them because it usually does not bring much improvement to performance [43].

4.2 Models development

The model development usually consists of model building, training, and validation. The development stage of the embedding component is merely a fine-tuning and validation of a model on a specific dataset, such as "Harry Potter." It is tuned to generate better numerical representations of sentences.

The development of the intent classifier needs more steps because it cannot use a pre-trained model. It requires configuring an architecture that best classifies user utterances and training it from scratch.

4.2.1 Building an architecture

Making adjustments to a model needs careful thinking about the architecture, the number of neurons in each layer, which activation function to use, and other details. The process is laborious and often requires many experiments. With every data update, an architecture should be reviewed and, in some cases, accustomed. For example, the intent classifier would have a different number of neurons in the output layer if the number of intents is changed. In some cases, the number of neurons in hidden layers must be adjusted to work adequately with a different number of labels. Based on the results of the experiments, a simple two-layer neural network classifier provides excellent results regardless of the number of intents and variations (the architecture was evaluated on datasets containing 7, 20, and 150 intents).

4.2.2 Hyperparameter tuning

Another essential part of model development is the tuning of hyperparameters. The process includes selecting such parameters as learning rate, batch size, weight decay, number of iterations, and many others. During the training of the intent classifier, the parameters were chosen using the grid search strategy. However, the technique is computationally demanding, so the hyperparameters of the embedding model remain similar to the default ones, which work reasonably well.

During hyperparameter tuning, it is essential to keep track of the best parameters and models. The embedding model is computationally expensive to train, so storing trained models and tracking respective hyperparameters for future usage is cost and time-efficient.

4.2.3 Model validation

The model validation procedure evaluates a model's reaction to previously unseen data. The datasets used in the intent detection module are divided into training, validation, and test sets. A test set simulates previously unseen data and contains different values of entities than training or validation sets. For example, the variation of intent *definition*, *"what does 'tertiary' mean"* belongs to the training set, while *"what does 'assiduous' mean"* is in the test set. These sentences represent the same variation, and the only difference is the entity value.

The validation is often used to adjust the model's parameters. In this work, a validation set was also used to select the threshold for filtering out-of-scope sentences, as was described in subsection 3.3.5.

During the validation, it is important to use suitable metrics. The k-fold cross-validation technique is often used even with a relatively large dataset because it provides a less biased estimation of a model's performance on the unseen data [44]. I used 3-fold cross-validation to evaluate the intent classifier on the CLINC150 in-scope dataset. The choice of k was influenced by the size of the training and test sets described in subsection 3.3.4.

Fold	all-MiniLM-L6-v2	all-roberta-large-v1
1	0.96	0.97
2	0.96	0.97
3	0.96	0.97
average	0.96	0.97

Table 4.1 3-fold cross-validation of the intent classifier on the concatenation of CLINC150 "in-scope" training and test sets.

Table 4.1 shows slightly higher accuracy than the one shown in Table 3.7. It might be influenced by the fact that the training and test sets contain different clusters of entities that tend to appear in distinct sentences of the same split. For example, variations of the intent *transfer* contain entity *"50$"* in the training set and *"20$"* in different sentences from the test set. Concatenation of this type of data leads to a slightly more optimistic accuracy estimation.

4.3 Models deployment

The goal of the deployment stage is to make a machine learning model available in production so that different systems can extract a model's predictions and other functionality. The deployment is an essential part of a machine learning life cycle because models and data are expected to be updated regularly.

The development uses different tools and environments than deployment because they serve different purposes. The stages might even run on separate hardware.

Machine learning models are usually a part of a larger software project, and different systems and applications can use them. The embedding model can be used not just by the intent detection module but also by, for example, the entity extractor. So it is better to deploy models independently from the rest of the program.

A detachment of machine learning models has several main advantages:

- Different systems can share a model's functionality.

- It is easier to add new functionalities and manage the whole machine learning life cycle.
- It is easier to monitor a model to ensure its proper operation.

4.3.1 Models as web services

For different software systems to be able to access machine learning models, a code needs to function as a web service, which runs on hardware and listens for requests at a particular port.

A web server is accessed via Representational State Transfer (REST) API. API is a general set of rules that defines how programs communicate with each other [45]. Additionally, REST defines an architectural style for an API that uses HTTP requests for exchanging data.

Deploying a model as a web server requires the creation of endpoints. An *endpoint* is a URL that provides the location from which an API can access resources on a server. They are points of contact between two systems, and for the main application to access a resource on a server, it must send a request to a specific URL. The most common HTTP request methods in API requests are: GET, POST, DELETE, and PUT.

Many web frameworks exist for developing REST API. For Python's programming language, the popular choices are Flask or FastApi. The latter is used in the intent detection module because it is easier to use and generally faster.

After creating endpoints, the application containing a model is containerized using Docker. It is a process of isolating an application from its environment to allow running it on different platforms without complications. Citing the official Docker website: *"Containers are a standardized unit of software that allows developers to isolate their app from its environment, solving the 'it works on my machine' headache [46]."*

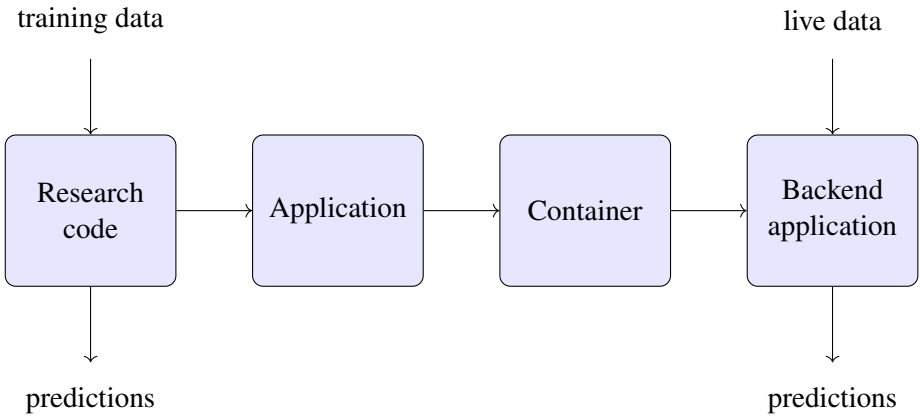


Figure 4.2 Stages of model deployment.

For other software systems to be able to access the functionality remotely, an application must have a public URL. One of the simplest ways to deploy a machine learning model to the internet is by running it on a cloud computing platform. There are currently three primary cloud providers that offer similar functionality: Amazon Web Services (AWS), Microsoft Azure, and Google Cloud [47]. The intent detection module runs on AWS because most of the other projects in the target company use AWS, so the environment is easier to maintain.

The intent detection module contains the embedding and intent classification components. They both use models that must be trained, validated, and persisted for making predictions. The simplest way of storing models and other required files is using local storage, but it is better to use cloud storage, such as AWS Simple Storage Service (S3), Minio, Azure Blob Storage,

or Google Cloud Storage. Cloud storage allows sharing models across different applications and systems. The intent detection module uses AWS S3 because it is consistent with a cloud provider.

Deploying models as web services is summarized in Figure 4.2. The process has several issues to consider:

- This type of deployment strategy does not handle multi-model serving well. When a system uses several models, it must load them on demand making the inference less computationally efficient.
- It is hard to monitor the model's operation.

4.3.2 Deployment frameworks

The deployment of machine learning models is a relatively laborious process that requires many manual steps. Because of it, I decided to use a framework called *TorchServe*. It is an open-source production framework created by PyTorch. TorchServe standardizes a model serving and makes model deployment and management in the production environment easier [48]. The framework provides a set of features needed for model serving, including a server, API specifications, metrics, and logging tools.

TorchServe server does not need any initial data to run. Models and other required files can be uploaded there later using the POST command. TorchServe can download and unpack them automatically.

In order to use a model on a server, a model's handler needs to be defined. A *handler* is a program that describes the model's inference and data processing phases. TorchServe provides standard handlers for common tasks, such as text classification. A custom handler can be implemented as well, and it should define the following functionalities [49]:

- **Initialization** defines how to load a model and create an instance.
- **Preprocessing** describes how data are transformed into a format that a model can understand.
- **Inference** is the primary function that defines the model's prediction step.
- **Postprocessing** defines how to modify the resulting data before sending a response.

In order to upload a new model to a server, an archive containing the model's files and handler must be created. After packaging, it can be stored either locally or in cloud storage. After that, the POST request is sent to a running server. The framework automatically downloads and registers a model and makes it ready to be used and managed using a standard API.

The following properties make TorchServe suitable for the intent detection module:

- **Multi-model serving**. TorchServe handles the serving of multiple models. Each of them is ready to be used when needed.
- **Low latency**. Serving on TorchServe is efficient even when multiple models are running.
- **Easier model management**. TorchServe API allows getting predictions, scaling workers, and model management.
- **Dynamical scaling of workers**. Workers can be dynamically added or removed from a specified model.

Another helpful framework is called *MLflow*. MLflow is an open-source platform designed to manage a model throughout the whole machine learning life cycle. It is integrated with popular machine learning tools, libraries, and frameworks, including TorchServe. It is used not only for deployment but also for automatic tracking of experiments [50].

The framework can be easily integrated with already existing machine learning code. Some variables, usually hyperparameters and metrics, can be specially marked for logging. MLflow provides an opportunity to reproduce variables that provide the best results. I experimented with MLflow but did not integrate it into the final product. MLflow is more beneficial for more complicated models where it is easier to lose track of the experiments.

There are many other frameworks for model deployment, for example, BentoML[7], Cortex[8], or Tensorflow Serving[9]. I will not investigate them further because these frameworks have similar functionality as TorchServe.

4.3.3 Deployment to Amazon EC2

The final application uses TorchServe as the main framework for model management. The TorchServe server runs on an instance on Amazon Elastic Compute Cloud (EC2) configured as follows:

- Ubuntu server 20.04 LTS,
- instance type t2.xlarge (4 CPUs, 16 GiB memory),
- 30 GiB storage.

After the initial configuration of an instance, a TorchServe server is started without any models. The intent detection module adds the embedding model during the initialization. A default encoder is MiniLM, but the embedding model is configurable, so any bi-encoder from the Sentence-Transformers library will work. The server downloads a model from AWS S3, registers it, and invokes a handler when a model's prediction is requested. The embedding model's handler expects a list of sentences as an input, and the intent classifier's handler expects a vector.

Machine learning components of the intent detection module are implemented in Python because most of the libraries mentioned in this work are available only in this programming language. The main libraries for development are SentenceTransformers[10], PyTorch[11], and Transformers[12]. I used PyTorch to develop neural networks despite other available options (Scikit-learn[13], Tensorflow[14]) because of the compatibility with TorchServe.

[7] https://github.com/bentoml/BentoML
[8] https://www.cortex.dev/
[9] https://www.tensorflow.org/tfx/guide/serving
[10] https://www.sbert.net/
[11] https://pytorch.org/
[12] https://huggingface.co/docs/transformers/index
[13] https://scikit-learn.org/stable/
[14] https://www.tensorflow.org/

Chapter 5

Intent detection module

The intent detection module consists of two separate applications. The first is a server that manages machine learning components, and the second is the main application that provides an API for intent recognition. Such a division offers other software systems the opportunity to use machine learning models and share their functionality.

5.1 Module's architecture for multiple applications

In order to classify user utterances, the intent detection module must maintain a set of potential intents and variations. Depending on the use case, these intents can belong to different topics. For example, a banking application could have intents related to transactions, payments, and card operations, while a cooking assistant would have a set of topics such as recipes, diets, meal suggestions, and others. Rather than predefining templates for various applications, the intent detection module allows users to add custom intents and variations. In this work, we call these users *clients*. Clients can use the module to classify user utterances related to their use case.

5.1.1 Machine learning components

Each client manages an independent application with a custom set of intents and needs a separate neural network-based classifier. All classifiers share architecture and hyperparameters (same as described in subsection 3.2.2), so the module only needs to store their weights.

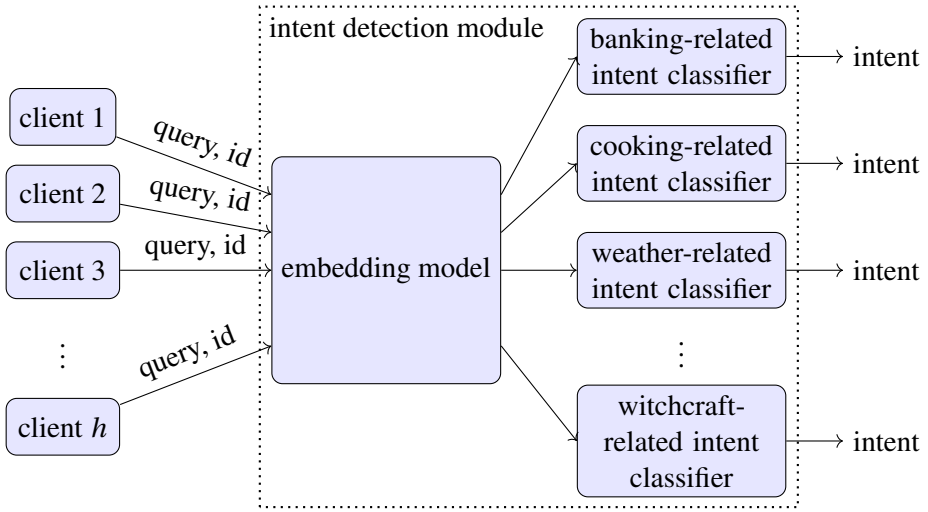


Figure 5.1 The module's design for h independent applications. The variable *query* represents a user message, and the *id* is used to identify a suitable application (classifier).

Clients use separate classifiers but share the embedding model. The model is pre-trained and fine-tuned on a large amount of data and captures general texts and intents well. If a user wants to discuss topics that use unusual vocabulary, such as "Harry Potter" books, as discussed in subsection 2.2.3, the embedding model can be fine-tuned.

Figure 5.1 illustrates the design of the module for h independent applications. It uses two types of machine learning components: the embedding model and the intent classifier. The embedding model is shared by h different clients, but each of them uses a separate intent classifier. The whole application contains $h+1$ machine learning models, so an efficient multi-model serving strategy is necessary.

The module usually starts without any loaded models. However, a default intent classifier is created for demonstration purposes. It is trained on the CLINC150 dataset and runs on a server with the embedding model. The module automatically generates and adds other classifiers when a client creates a dataset.

5.1.2 The core component

An application that connects all components of the intent detection module is called the *core component*. It is the main program that implements several functionalities:

- an API for managing datasets and getting predictions,
- wrappers for communication with the TorchServe server and AWS S3,
- a front-end application for users, and
- a training module where models are trained and packaged.

The core component implements two classes for machine learning elements and a wrapper for communicating with a server. Classes for machine learning models have two basic functions:

- `archive()` collects required files, such as saved weights, vocabulary, handler, and other data, packages them into an archive, and uploads them to AWS S3.
- `train()` trains and stores the model.

The module implements a wrapper that provides an interface for communication with the server. It manages models served on Amazon EC2 using six main functions:

- `register(model)` deploys a new model to the server.
- `is_registered(id)` checks if a model is running on the server.
- `add_workers(id, num_workers)` adds or removes workers from a model.
- `get_prediction(id, query)` returns a prediction of a particular model.
- `unregister(id)` removes a model from the server.
- `unregister_all()` removes all running models from the server.

5.1.3 Module's arrangement

The core application is a program that maintains the module's components, communicates with clients, and manages models on the server. It runs on an EC2 instance as a separate program, which communicates via HTTP with other applications. Figure 5.2 illustrates the resulting module's architecture and components arrangement.

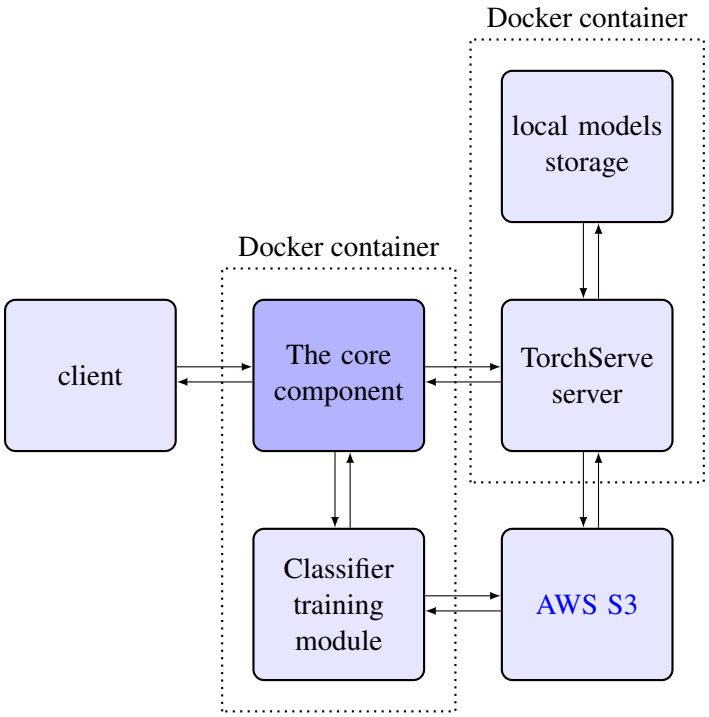


Figure 5.2 The intent detection module's architecture.

5.2 Implementation

5.2.1 Initialization of the module

The intent detection module registers the embedding model during the initialization. The embedding model is configurable with MiniLM as a default one. The following pseudocode describes the initialization procedure:

Pseudocode

```
1:  global server, embedding_model
2:  function INITIALIZE(dataset)
3:      server = create_server_wrapper()
4:      embedding_model = create_embedding_model()
5:      if dataset ≠ None then
6:          embedding_model.train(dataset) # fine-tuning
7:      end if
8:      embedding_model.archive()
9:      server.register(embedding_model)
10: end function
```

The first part of the algorithm initializes the variables *server* and *embedding_model*. The variable *server* is an object of a TorchServe API wrapper. The *embedding_model* object is responsible for creating, fine-tuning, storing, and packaging bi-encoder models. It is updated only during the initialization, but it is possible to add endpoints for fine-tuning in the future.

5.2.2 Client management

The application creates and automatically deploys a new neural network-based classifier when a user adds or modifies a dataset. A model needs to train from scratch because users can add or remove intents (classes), modifying the number of output neurons. The following pseudocode describes the update of an application:

Pseudocode

```
1: global server
2: function UPDATE(id, data)
3:     classifier = create_classifier()
4:     classifier.train(data)
5:     if server.is_registered(id) == True then
6:         server.unregister(id)
7:     end if
8:     server.register(classifier)
9: end function
```

The pseudocode also covers the registration of a new client because, in either case (registration of a new or update of an old one), a model trains from scratch. The only difference between registration and updating is checking if a client is already registered. If a model with a particular *id* is already present on the server, the registration of a model with the same name will result in an error. Therefore, an old model must be removed first.

In a real application, a model is not available for prediction during its management on the server. So a prediction call to an application that is currently being removed from the server might result in an error. This issue might be addressed by, for example, setting a timeout. Luckily, the intent classifier is small, and uploading it to the server takes only several milliseconds.

It is worth mentioning that the module must identify a classifier that needs to be updated, so the function requires an argument called *id*. For simplicity, the program assumes that the *id* is known and provided by a client.

5.2.3 Predictions

The following pseudocode describes how a client requests the module to classify an intent:

Pseudocode

```
1: server, embedding_model
2: function GET_INTENT(id, query)
3:     embedding = server.get_prediction(embedding_model.id, query)
4:     intent = server.get_prediction(id, embedding)
5:     return intent
6: end function
```

The algorithm first transforms an utterance into a vector by sending a request to the embedding model that runs on the server. Then it requests an intent classifier to assign a label to an embedding.

Function `get_prediction()` requests a specific URL that invokes the machine learning pipeline defined in a Torch handler. TorchServe provides the required API for invoking the inference and handling the errors.

5.3 Deploying the module in production

5.3.1 Application programming interface

The Intent detection module implements the following endpoints:

- The `api/clients` endpoint accepts GET, POST, and DELETE methods. It registers, unregisters clients, and provides basic information for each application. The POST method expects user data in a dictionary, where a sentence is a key and intent is a value. The program replaces an existing dataset with a new one and then trains a classifier. The training is usually fast, and it does not last more than a few seconds.

- The `api/intent` endpoint accepts only the GET method. It takes a client id and a query as an input and returns a predicted intent.

The application runs on a web server provided by Uvicorn. It listens on a socket and passes requests to FastAPI. The following code shows an example of a request acceptable by the application that runs locally on port 5555:

```
curl -X 'GET' \
'http://127.0.0.1:5555/api/intent?query=hello%20world' \
-H 'accept: application/json'
```

The request asks the intent detection module to classify the sentence *"Hello world."* When a user does not provide a client id, the application uses a default classifier, which, given the sentence, predicts the intent *greeting*.

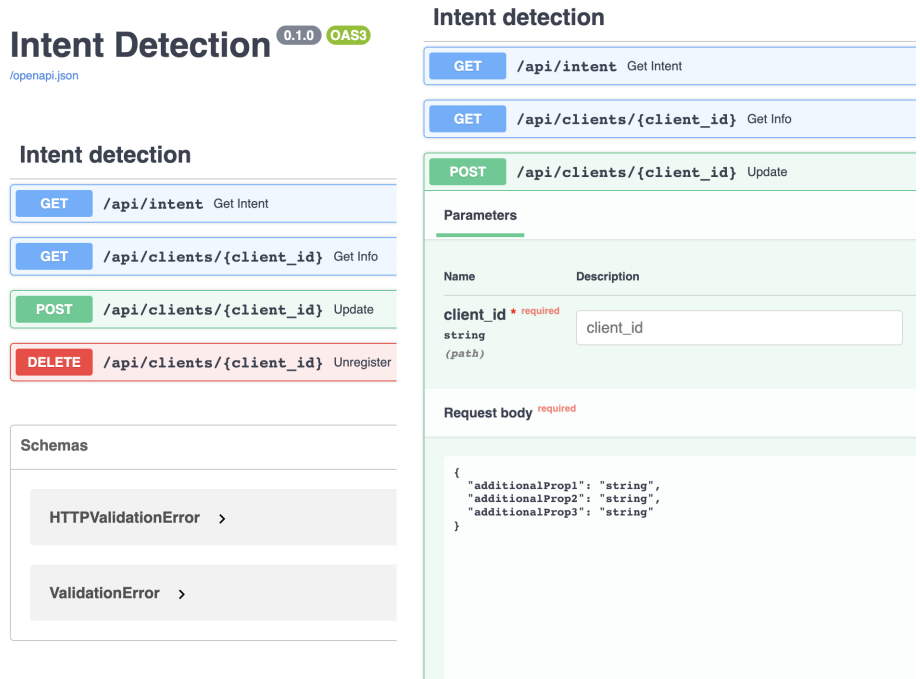


Figure 5.3 An interactive API documentation in Swagger User Interface (UI).

Some requests, for example, posting an update, are more complicated and require data in a particular form. Therefore, documentation that explains the usage of an API was generated

using Swagger UI[1]. It provides automatic interactive documentation, which is demonstrated in Figure 5.3.

5.3.2 AWS deployment

In order to make the intent detection module available for different software systems, a code is uploaded to an EC2 instance. Machine learning components and the core application run on the same instance. However, components can be just as well uploaded to different instances using the same procedure. For example, the core component can run on a serverless computing platform, such as AWS Lambda.

As was described in Chapter 4, the final application is containerized using Docker. The command *docker build* creates an image that collects the code, dependencies, configuration, and required files and creates an executable package that is OS agnostic.

The image is uploaded to the AWS Elastic Container Registry (ECR) service that allows storing, sharing, and deploying container images. From there, an EC2 instance pulls the image and executes it locally. The steps are summarized in Figure 5.4.

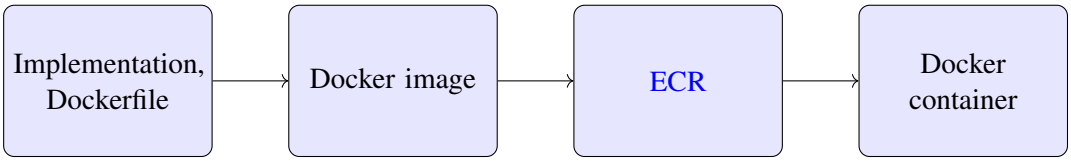


Figure 5.4 The main steps of the intent detection module's deployment.

5.3.3 Speed of the application

One of the essential criteria in deployment is the latency of the whole system. Table 5.1 shows the module's request processing time.

	all-MiniLM-L6-v2	**all-roberta-large-v1**
Inference time [ms]	33	120

Table 5.1 Approximate inference time of getting a prediction. The module's container is deployed to an AWS EC2 instance with a default classifier trained on the CLINC150 dataset. The evaluation was performed on the same instance, so the network latency does not influence the measurement.

The resulting inference time is slower than the combined encoding and intent classification time by approximately 20 ms because the module has to do additional preprocessing and post-processing steps. For example, a tensor (a type used for embeddings) has to be serialized to JSON format before being sent as a response. Then it is converted to a tensor again when received by the intent classifier.

The inference of the module with the S-ROBERTA encoder is approximately four times slower than one with MiniLM. Moreover, S-ROBERTA's size is 1.2 GB, while MiniLM's is just 79.7 MB. MiniLM is approximately 17 times smaller and four times faster and, therefore, is used as a default encoder in the intent detection module.

[1] https://swagger.io/

Chapter 6

Conclusion

Intent detection is a text classification task that represents a fundamental component in conversational agents and chatbots. The aim of this work was to develop an intent detection module able to classify user utterances using a predefined set of intents. The proposed solution consists of the following parts: sentence embeddings, intent classification, and deployment strategy.

Any machine learning-based classifier works with numerical inputs. The most useful numerical representation of sentences is one that preserves their meaning. Modern state-of-the-art algorithms that deal with this problem are transformer-based models BERT and its derivatives. Three algorithms for semantic textual similarity were evaluated in this work: S-ROBERTA, MiniLM, and the Sent2Vec algorithm serving as a reference model. The models were evaluated on two different datasets: the STSB and "Harry Potter," with the Pearson correlation coefficient measuring the relationship between predictions and annotated similarity scores. On the STSB dataset, the Sent2Vec algorithm achieved Pearson correlation coefficient 0.76 and the pre-trained transformer-based models 0.83 (MiniLM) and 0.84 (S-ROBERTA). Fine-tuning using STSB training data improved performance only by 2-4%. On the "Harry Potter" dataset, the Sent2Vec achieved 0.85 and transformer-based models 0.91 (MiniLM) and 0.84 (S-ROBERTA). The fine-tuning significantly improved performance of transformer models to 0.99.

While S-ROBERTA offers a better language understanding, the distilled MiniLM provides comparable performance and is four times faster. A smaller and low-latency model is preferable in a conversational assistant because the intent detection module is usually a part of a larger system, so it should be as fast as possible. Therefore, MiniLM is used as a default encoder in the final product.

The intent detection system can be implemented using a semantic similarity-based approach (unsupervised), which returns the intent of the most similar sentence in the training data and does not require any additional learning. On the other hand, the supervised approach uses an additional neural network for classification, in which weights are learned from training data. The comparison of intent classification accuracy was performed on publicly available datasets SNIPS and CLINC150 and on an internal dataset dealing with appointment booking queries. Both approaches performed well on SNIPS (92-96%), but the supervised approach performed 4-8% better on the in-scope CLINC150, achieving 94-96% accuracy. Neither approach was able to categorize out-of-scope examples (accuracy was lower than 50%) and, therefore, a threshold-based solution was implemented for filtering such sentences. It improved the out-of-scope accuracy to 80-85% while decreasing in-scope accuracy only by 2-5% to 85-90%.

The developed module was compared with the existing solution based on Sent2Vec embeddings, and it achieves similar results on SNIPS, approximately 10-12% better results on CLINC150, and 15% better results on an internal dataset.

The MiniLM-based supervised approach has a good performance, it is generally faster and

can deal with entities better than the unsupervised MiniLM, so it was chosen as a base method for the final product.

This work also focused on the development and deployment of the module and its machine learning components. The final application allows users to add custom intents and variations. Each of them needs a separate neural network for classification, so in order to manage multiple machine learning components, the TorchServe framework was chosen as the preferred serving strategy. The rest of the application communicates with it via HTTP requests. The final application runs on an Amazon EC2 instance and can be accessed via REST API.

6.1 Future work

The intent detection module achieves good results, but the application can be modified to work even better in the future. The following modifications can improve the proposed solution:

- Advanced spelling correction of user utterances. The application is sensitive to errors occurring in sentences and would benefit from an additional machine learning-based preprocessing layer.
- Automatic modification of the intent classifier's architecture and hyperparameters during training. The proposed supervised architecture might not work well with some data (for example, when there are too many or too few intents), so some automatic adjustments (number of hidden layers, neurons in each layer, learning rate, and others) based on data might mitigate possible issues.
- Analysis of the most problematic entities. As briefly discussed in Chapter 3, the module does not perform at its best when dealing with sentences containing entities, and it might be helpful to detect some of them during the preprocessing step. It would require more annotated datasets.

Bibliography

[1] Karen Sparck Jones. "Natural Language Processing: A Historical Review". In: *Current Issues in Computational Linguistics: In Honour of Don Walker*. Ed. by Antonio Zampolli, Nicoletta Calzolari, and Martha Palmer. Dordrecht: Springer Netherlands, 1994, pp. 3–16. ISBN: 978-0-585-35958-8. DOI: 10.1007/978-0-585-35958-8_1. URL: https://doi.org/10.1007/978-0-585-35958-8_1 (visited on 04/08/2022).

[2] Prashant Johri et al. "Natural Language Processing: History, Evolution, Application, and Future Work". In: Jan. 1, 2021, pp. 365–375. ISBN: 9789811597114. DOI: 10.1007/978-981-15-9712-1_31.

[3] Jiao Liu, Yanling Li, and Min Lin. "Review of Intent Detection Methods in the Human-Machine Dialogue System". In: *Journal of Physics: Conference Series* 1267.1 (July 2019). Publisher: IOP Publishing, p. 012059. ISSN: 1742-6596. DOI: 10.1088/1742-6596/1267/1/012059. URL: https://doi.org/10.1088/1742-6596/1267/1/012059 (visited on 02/23/2022).

[4] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *arXiv:1810.04805 [cs]* (May 24, 2019). arXiv: 1810.04805. URL: http://arxiv.org/abs/1810.04805 (visited on 03/03/2022).

[5] Shusen Liu et al. "Visual Exploration of Semantic Relationships in Neural Word Embeddings". In: *IEEE Transactions on Visualization and Computer Graphics* 24.1 (Jan. 2018). Conference Name: IEEE Transactions on Visualization and Computer Graphics, pp. 553–562. ISSN: 1941-0506. DOI: 10.1109/TVCG.2017.2745141.

[6] Chunjie Luo et al. "Cosine Normalization: Using Cosine Similarity Instead of Dot Product in Neural Networks". In: *arXiv:1702.05870 [cs, stat]* (Oct. 22, 2017). arXiv: 1702.05870. URL: http://arxiv.org/abs/1702.05870 (visited on 03/02/2022).

[7] Tomas Mikolov et al. "Efficient Estimation of Word Representations in Vector Space". In: *arXiv:1301.3781 [cs]* (Sept. 6, 2013). arXiv: 1301.3781. URL: http://arxiv.org/abs/1301.3781 (visited on 03/02/2022).

[8] Matteo Pagliardini, Prakhar Gupta, and Martin Jaggi. "Unsupervised Learning of Sentence Embeddings using Compositional n-Gram Features". In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)* (2018), pp. 528–540. DOI: 10.18653/v1/N18-1049. arXiv: 1703.02507. URL: http://arxiv.org/abs/1703.02507 (visited on 03/02/2022).

[9] Piotr Bojanowski et al. "Enriching Word Vectors with Subword Information". In: *arXiv:1607.04606 [cs]* (June 19, 2017). arXiv: 1607.04606. URL: http://arxiv.org/abs/1607.04606 (visited on 03/02/2022).

[10] Dan Jurafsky and James H. Martin. *Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition, 2nd Edition*. Prentice Hall series in artificial intelligence. Prentice Hall, Pearson Education International, 2009. ISBN: 9780135041963. URL: https://www.worldcat.org/oclc/315913020.

[11] M. Schuster and K.K. Paliwal. "Bidirectional recurrent neural networks". In: *IEEE Transactions on Signal Processing* 45.11 (Nov. 1997). Conference Name: IEEE Transactions on Signal Processing, pp. 2673–2681. ISSN: 1941-0476. DOI: 10.1109/78.650093.

[12] Y. Bengio, Patrice Simard, and Paolo Frasconi. "Learning long-term dependencies with gradient descent is difficult". In: *IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council* 5 (Feb. 1, 1994), pp. 157–66. DOI: 10.1109/72.279181.

[13] Junyoung Chung et al. "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling". In: *arXiv:1412.3555 [cs]* (Dec. 11, 2014). arXiv: 1412.3555. URL: http://arxiv.org/abs/1412.3555 (visited on 03/03/2022).

[14] Alec Radford and Karthik Narasimhan. "Improving Language Understanding by Generative Pre-Training". In: *undefined* (2018). URL: https://www.semanticscholar.org/paper/Improving-Language-Understanding-by-Generative-Radford-Narasimhan/cd18800a0fe0b668a1cc19f2ec95b5003d0a5035 (visited on 05/16/2022).

[15] Lu Peng et al. "Exploiting Model-Level Parallelism in Recurrent Neural Network Accelerators". In: *2019 IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. 2019 IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC). Oct. 2019, pp. 241–248. DOI: 10.1109/MCSoC.2019.00042.

[16] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. "Neural Machine Translation by Jointly Learning to Align and Translate". In: *arXiv:1409.0473 [cs, stat]* (May 19, 2016). arXiv: 1409.0473. URL: http://arxiv.org/abs/1409.0473 (visited on 04/19/2022).

[17] Daniel W. Otter, Julian R. Medina, and Jugal K. Kalita. "A Survey of the Usages of Deep Learning in Natural Language Processing". In: *arXiv:1807.10854 [cs]* (Dec. 21, 2019). arXiv: 1807.10854. URL: http://arxiv.org/abs/1807.10854 (visited on 03/03/2022).

[18] Ashish Vaswani et al. "Attention Is All You Need". In: *arXiv:1706.03762 [cs]* (Dec. 5, 2017). arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762 (visited on 03/03/2022).

[19] Liyuan Liu, Jialu Liu, and Jiawei Han. "Multi-head or Single-head? An Empirical Comparison for Transformer Training". In: *arXiv:2106.09650 [cs]* (June 17, 2021). arXiv: 2106.09650. URL: http://arxiv.org/abs/2106.09650 (visited on 04/22/2022).

[20] Anna Rogers, Olga Kovaleva, and Anna Rumshisky. "A Primer in BERTology: What we know about how BERT works". In: *arXiv:2002.12327 [cs]* (Nov. 9, 2020). arXiv: 2002.12327. URL: http://arxiv.org/abs/2002.12327 (visited on 04/19/2022).

[21] Yonghui Wu et al. "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation". In: *arXiv:1609.08144 [cs]* (Oct. 8, 2016). version: 2. arXiv: 1609.08144. URL: http://arxiv.org/abs/1609.08144 (visited on 04/22/2022).

[22] Nils Reimers and Iryna Gurevych. “Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks”. In: *arXiv:1908.10084 [cs]* (Aug. 27, 2019). arXiv: 1908.10084. URL: http://arxiv.org/abs/1908.10084 (visited on 03/04/2022).

[23] Yinhan Liu et al. “RoBERTa: A Robustly Optimized BERT Pretraining Approach”. In: *arXiv:1907.11692 [cs]* (July 26, 2019). arXiv: 1907.11692. URL: http://arxiv.org/abs/1907.11692 (visited on 03/15/2022).

[24] *sentence-transformers/all-roberta-large-v1 · Hugging Face*. URL: https://huggingface.co/sentence-transformers/all-roberta-large-v1 (visited on 03/22/2022).

[25] Wenhui Wang et al. “MiniLM: Deep Self-Attention Distillation for Task-Agnostic Compression of Pre-Trained Transformers”. In: *arXiv:2002.10957 [cs]* (Apr. 5, 2020). arXiv: 2002.10957. URL: http://arxiv.org/abs/2002.10957 (visited on 03/15/2022).

[26] *sentence-transformers/all-MiniLM-L6-v2 · Hugging Face*. URL: https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2 (visited on 03/22/2022).

[27] Tomas Brich. “Semantic Sentence Similarity for Intent Recognition Task”. MA thesis. Czech technical university in Prague, May 2018. URL: https://dspace.cvut.cz/handle/10467/77029 (visited on 05/06/2022).

[28] Po-Sen Huang et al. “Learning deep structured semantic models for web search using clickthrough data”. In: *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management - CIKM ’13*. the 22nd ACM international conference. San Francisco, California, USA: ACM Press, 2013, pp. 2333–2338. ISBN: 978-1-4503-2263-8. DOI: 10.1145/2505515.2505665. URL: http://dl.acm.org/citation.cfm?doid=2505515.2505665 (visited on 04/11/2022).

[29] Yun-Nung Chen, Dilek Hakkani-Tür, and Xiaodong He. “Zero-shot learning of intent embeddings for expansion by convolutional deep structured semantic models”. In: *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). ISSN: 2379-190X. Mar. 2016, pp. 6045–6049. DOI: 10.1109/ICASSP.2016.7472838.

[30] Buddhika Kasthuriarachchy et al. “Pre-trained Language Models with Limited Data for Intent Classification”. In: *2020 International Joint Conference on Neural Networks (IJCNN)*. 2020 International Joint Conference on Neural Networks (IJCNN). ISSN: 2161-4407. July 2020, pp. 1–9. DOI: 10.1109/IJCNN48605.2020.9207121.

[31] Wenda Chen, Jonathan Huang, and Mark Hasegawa-Johnson. “Utterance-level Intent Recognition from Keywords”. In: *arXiv:2009.08064 [cs, eess]* (Sept. 17, 2020). arXiv: 2009.08064. URL: http://arxiv.org/abs/2009.08064 (visited on 04/11/2022).

[32] Andrea A. Lunsford and Karen J. Lunsford. “"Mistakes Are a Fact of Life": A National Comparative Study”. In: *College Composition and Communication* 59.4 (2008). Publisher: National Council of Teachers of English, pp. 781–806. ISSN: 0010-096X. URL: https://www.jstor.org/stable/20457033 (visited on 03/14/2022).

[33] Lichao Sun et al. “Adv-BERT: BERT is not robust on misspellings! Generating nature adversarial samples on BERT”. In: *arXiv:2003.04985 [cs]* (Feb. 27, 2020). arXiv: 2003.04985. URL: http://arxiv.org/abs/2003.04985 (visited on 04/11/2022).

[34] Daria Ozerova. "Semantic sentence similarity". Unpublished. In: (Jan. 26, 2022). URL: https://github.com/ozerodar/semantic_sentence_similarity/blob/main/sentence_similarity.pdf (visited on 04/11/2022).

[35] *In-Depth Guide Into Chatbots Intent Recognition in 2022*. Mar. 24, 2021. URL: https://research.aimultiple.com/chatbot-intent/ (visited on 03/14/2022).

[36] Qian Chen, Zhu Zhuo, and Wen Wang. "BERT for Joint Intent Classification and Slot Filling". In: *arXiv:1902.10909 [cs]* (Feb. 28, 2019). arXiv: 1902.10909. URL: http://arxiv.org/abs/1902.10909 (visited on 04/27/2022).

[37] Alice Coucke et al. "Snips Voice Platform: an embedded Spoken Language Understanding system for private-by-design voice interfaces". In: *arXiv:1805.10190 [cs]* (Dec. 6, 2018). version: 3. arXiv: 1805.10190. URL: http://arxiv.org/abs/1805.10190 (visited on 05/02/2022).

[38] Stefan Larson et al. "An Evaluation Dataset for Intent Classification and Out-of-Scope Prediction". In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. EMNLP-IJCNLP 2019. Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 1311–1316. DOI: 10.18653/v1/D19-1131. URL: https://aclanthology.org/D19-1131 (visited on 04/22/2022).

[39] Libo Qin et al. "A Stack-Propagation Framework with Token-Level Intent Detection for Spoken Language Understanding". In: *arXiv:1909.02188 [cs]* (Sept. 4, 2019). version: 1. arXiv: 1909.02188. URL: http://arxiv.org/abs/1909.02188 (visited on 04/24/2022).

[40] *Machine Learning Life Cycle: Top 3 Components*. Deepchecks. June 30, 2021. URL: https://deepchecks.com/understanding-the-machine-learning-life-cycle/ (visited on 03/05/2022).

[41] Xiang Zhang, Junbo Zhao, and Yann LeCun. "Character-level Convolutional Networks for Text Classification". In: *Advances in Neural Information Processing Systems*. Vol. 28. Curran Associates, Inc., 2015. URL: https://proceedings.neurips.cc/paper/2015/hash/250cf8b51c773f3f8dc8b4be867a9a02-Abstract.html (visited on 03/29/2022).

[42] Steven Y. Feng et al. "A Survey of Data Augmentation Approaches for NLP". In: *arXiv:2105.03075 [cs]* (Dec. 1, 2021). arXiv: 2105.03075. URL: http://arxiv.org/abs/2105.03075 (visited on 03/29/2022).

[43] Jason Wei and Kai Zou. "EDA: Easy Data Augmentation Techniques for Boosting Performance on Text Classification Tasks". In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. EMNLP-IJCNLP 2019. Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 6382–6388. DOI: 10.18653/v1/D19-1670. URL: https://aclanthology.org/D19-1670 (visited on 03/29/2022).

[44] Sebastian Raschka. *Model Evaluation, Model Selection, and Algorithm Selection in Machine Learning*. arXiv:1811.12808. type: article. arXiv, Nov. 10, 2020. arXiv: 1811.12808[cs,stat]. URL: http://arxiv.org/abs/1811.12808 (visited on 05/16/2022).

[45] *rest-apis*. Aug. 4, 2021. URL: https://www.ibm.com/uk-en/cloud/learn/rest-apis (visited on 04/01/2022).

[46] *Why Docker*. Docker. URL: https://www.docker.com/why-docker/ (visited on 04/01/2022).

[47] Scott Carey. *AWS vs Azure vs Google Cloud: What's the best cloud platform for enterprise?* Computerworld. Jan. 23, 2020. URL: https://www.computerworld.com/article/3429365/aws-vs-azure-vs-google-whats-the-best-cloud-platform-for-enterprise.html (visited on 03/31/2022).

[48] *torchserve-on-aws*. URL: https://catalog.us-east-1.prod.workshops.aws/workshops/04eb9f59-6d25-40c5-a828-67df58b85739/en-US/900 (visited on 04/14/2022).

[49] *6. Custom Service — PyTorch/Serve master documentation*. URL: https://pytorch.org/serve/custom_service.html (visited on 04/14/2022).

[50] *MLflow Tracking — MLflow 1.25.1 documentation*. URL: https://mlflow.org/docs/latest/tracking.html (visited on 04/14/2022).